动态语言技术

精品书廊

Broadview
www.broadview.com.cn



# Python源码剖析

## ——深度探索动态语言核心技术

陈儒　著

哲思自由软件社区　审校

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT…

LANZAROTE Caliente.com

Broadview www.broadview.com.cn 技术凝聚实力 专业创新出版



# Python源码剖析

## ——深度探索动态语言核心技术

作为主流的动态语言，Python不仅简单易学、移植性好，而且拥有强大丰富的库的支持。此外，Python强大的可扩展性，让开发人员既可以非常容易地利用C/C++编写Python的扩展模块，还能将Python嵌入到C/C++程序中，为自己的系统添加动态扩展和动态编程的能力。实践证明，在数据挖掘、图像处理、网络游戏，以及Web开发等领域的开发实践中，采用Python都能极大地提高开发的效率。

为了更好地利用Python语言，无论是使用Python语言本身，还是将Python与C/C++交互使用，深刻理解Python的运行原理都是非常重要的。本书以CPython为研究对象，在C代码一级，深入细致地剖析了Python的实现。本书不仅包括了对大量Python内置对象的剖析，更将大量的篇幅用于对Python虚拟机及Python高级特性的剖析。通过此书，读者能够透彻地理解Python中的一般表达式、控制结构、异常机制、类机制、多线程机制、模块的动态加载机制、内存管理机制等核心技术的运行原理，同时，本书所揭示的动态语言的核心技术对于理解其他动态语言，如JavaScript、Ruby等也有较大的参考价值。

### 本书的主要特点

- 一本深入剖析Python具体实现的著作
- 内容新鲜，采用最新的Python语言版本（V2.5）
- 大量的图表形象地展示Python内部的运作机理
- 在原理介绍的同时，带领读者一起动手对Python虚拟机进行改造
- 完整覆盖Python所有的核心议题，深刻揭示Python与C/C++之间如何互动

### 作者介绍


- 陈儒，计算机科学与工程专业硕士，问天（北京）信息技术有限公司技术负责人，致力于信息检索方向的研究与开发（*www.isoche.com*）。

  个人博客：Dynamic Life（*http://blog.csdn.net/balabalamerobert*）。


图书分类：程序设计

网上订购：www.dearbook.com.cn 第二书店·第一服务

策划编辑：周 筠
责任编辑：杨绣国
责任美编：胡文佳

ISBN 978-7-121-06874-4

9 787121 068744 >

定价：69.80元

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

# Python 源码剖析

——深度探索动态语言核心技术

陈 儒 著

哲思自由软件社区 审校

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

## 内 容 简 介

作为主流的动态语言，Python 不仅简单易学、移植性好，而且拥有强大丰富的库的支持。此外，Python 强大的可扩展性，让开发人员既可以非常容易地利用 C/C++编写 Python 的扩展模块，还能将 Python 嵌入到 C/C++程序中，为自己的系统添加动态扩展和动态编程的能力。

为了更好地利用 Python 语言，无论是使用 Python 语言本身，还是将 Python 与 C/C++交互使用，深刻理解 Python 的运行原理都是非常重要的。本书以 CPython 为研究对象，在 C 代码一级，深入细致地剖析了 Python 的实现。书中不仅包括了对大量 Python 内置对象的剖析，更将大量的篇幅用于对 Python 虚拟机及 Python 高级特性的剖析。通过此书，读者能够透彻地理解 Python 中的一般表达式、控制结构、异常机制、类机制、多线程机制、模块的动态加载机制、内存管理机制等核心技术的运行原理，同时，本书所揭示的动态语言的核心技术对于理解其他动态语言，如 Javascript、Ruby 等也有较大的参考价值。

本书适合于 Python 程序员、动态语言爱好者、C 程序员阅读。

未经许可，不得以任何方式复制或抄袭本书的任何部分。
版权所有，侵权必究。

**图书在版编目（CIP）数据**

Python 源码剖析：深度探索动态语言核心技术 / 陈儒 著.—北京：电子工业出版社，2008.6
ISBN 978-7-121-06874-4
I. P… Ⅱ.陈… Ⅲ.软件工具－程序设计 Ⅳ.TP311.56

中国版本图书馆 CIP 数据核字（2008）第 083441 号

责任编辑：杨绣国
印　　刷：北京智力达印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：31.75　字数：600 千字
印　　次：2008 年 6 月第 1 次印刷
印　　数：4 000 册　　定价：69.80 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

# 推荐序一

## 让我们做得更好

Python，我想已经不再是个陌生的词了，越来越多的人开始学习它，使用它，宣传它，甚至用它找到了工作。如果你了解 Python，那么我想问一下，你对它有多了解呢？它是一种什么语言？如何实现的？有哪些对象，它们是如何处理的？你了解 Python 的虚拟机吗？了解它的运行环境吗？其实作为初学者或只是使用者，你的确不必了解这么多细节的内容，但是探究事物的原理，分析底层细节却也是许多人成为高手、“老鸟”的原因，因为你知道别人不知道的东西，掌握了别人不了解的技术，这些内容使得你的见解、分析，甚至作品都可能超过别人。那么本书就向你提供了一个了解 Python 底层细节的机会，你可以沿着作者的思路和角度去体会 Python 的工作原理和底层的细节，一点一点地了解 Python 源码的精妙之处，有助于更好地掌握 Python 并编写出高质量的程序。

本书的内容深入到 Python 的方方面面，像 Python 的对象实现机制是如何用 C 来表现的；对象的特性是如何实现的；对象是如何管理的；不同对象，如 `int`、`str`、`dict`、`list` 等的处理；Python 的虚拟机框架、作用域的实现；运行时环境，pyc 文件，类机制等。还有一些高级话题，如内存管理，GIL（Global Interpreter Lock）与多线程，模块动态加载等。

在接触到本书之前，我已经在作者的 Blog 上见到过部分内容，那时已经被作者不懈的毅力和深厚的功力折服。说实话，由于经常接触 Python，对于原本熟悉的 C 语言也越来越陌生，更不要说去“啃”Python 的 C 代码了。而 Robert Chen 可以从源码中进行有条理的分析和整理，并终于出版此书。这不仅让人敬佩，更让广大的 Python 爱好者受益多多。因此，当出版社希望我为本书作一个推荐序时，我毫不犹豫地答应下来。

第一次见到 Robert Chen 还是在 CPUG 的一次会课上，那时 Robert Chen 给大家带来一个主题为“Python 作用域与名字空间”的讲座，让在座的 Pythoner 对 Python 的作用域机制有了更深入的理解，讲座效果非常好，讨论也很热烈。他从源码的角度讲述了 Python

的一些规则，使得大家的理解不再停留在形式上或规则上，而达到了本质或实现的层次，让我们“知其然，更知其所以然”。

如果你是一位热心的 Pythoner，想必会知道中文 Python 的邮件列表（*http://groups.google.com/group/python-cn*），从上面对有些问题的回复中，你会发现 Robert Chen 总是从源码及字节码实现的角度来回答问题，非常有说服力。因此当许多 Python 爱好者得知 Robert Chen 将出版此书时，都非常盼望，现在这本书终于出版，大家都深感庆幸！

本书不仅仅是高水平、高质量的一本书，纵观国内外与 Python 相关的书籍，它也是第一本从源码角度写作的书，所以意义非常。目前国内原创的 Python 书籍还不多，就我所知国内已经出版的几本 Python 方面的书尚不能满足读者需求，而本书应该不会让你失望。

不过本书应该不是面向初学者的书，因为它涉及了许多较深的内容和知识，建议读者应先掌握像 C 语言、数据结构、操作系统、编译原理等方面的基础知识，并且具备一定的编程经验，才能更好地理解书的内容。

再次感谢 Robert Chen 带给大家的这份礼物！

李迎辉

*limodou@gmail.com*

2008 年春

# 推荐序二

真的难以想象，Python 语言和社区能够发展得如此迅速。在我第一次使用 Python 完成我的项目时，它还不过是一个刚刚在开源社区中起步的新生儿，然后在各方面迅速推广，4 月 8 日，google 发布的 App Engine 更是让所有的开发人员眼中一亮。相信今后会有更多的开发人员投入到 Python 的技术领域中来。

记得在 2002 年时，我使用 Python 写了一套大规模的消息系统，几位同事分别使用 Python、Java 和 C 完成了一个异步二进制消息流的客户端和服务器。通过一系列测试，大家惊奇地发现 Python 以每秒一倍的数据处理量超过了 C 写的代码。后来，我的同事细心地查看了 Python 的源代码，发现了几种完全不同的操作系统调用方法，以及为提高性能而使用的技巧。这也是我第一次开始查看 Python 的源代码。最近的一次则是我在 xBayTable 中使用 asyncore 时，通过阅读 asyncore 的源代码排除了一个痛苦 bug，轻松地找到了问题的根源，很快就换了一种解决方案来继续我的工作。了解 Python 的源代码，我们能获得很多的好处：

- 使用 Python 方法提高自己的代码性能和功能；
- 快速地与文档结合，解决问题或是找出方法；
- 扩展 Python。

我常将所有的书分为口袋书、马桶书、枕头书。RobertChen 的《Python 源码剖析》，更多讲述的是 CPython 中的实现技术和方法。这可以让我们从不同的层面了解 Python 简洁背后的机理。我更推荐大家把它当做口袋书，在准备书写 Python 扩展前把它作为一本工具书，配合“Extending and Embedding the Python Interpreter”会让你更容易地完成你的工作。另一方面，当你想使用 Python 这种方法解决问题时，这本书也可以成为你的好伙伴，它让你更多更快地了解 Python 是怎么做的，从而做得“和 Python 一样”。当你对 Python 的一些问题百思不得其解时，这本书也许可以从不同的方面帮助你了解它最底层发生的故事。

最后，作为 Python 社区的长期参与者，希望更多的代码人不但能使用 Python 语言去完成自己的工作，也希望能有更多的人通过这本书成为 Python 语言的开发者，更希望中国有越来越多的 Python 开发者。

黄冬

2008 年 4 月于北京

下雨的深夜

# 推荐序三

非常高兴看到又一本原创的 Python 图书出版。

说起来，我和 Python 还算有一点缘分。在 2000 年的时候，一次非常偶然的机会我接触到 Python，当时网上的资料非常少，不知天高地厚的我竟冒失地接手了国内第一本引进版 Python 图书的合作翻译工作，往事不堪回首。记得当时经常有人问我 Python 能用来做什么……而我能举出来的例子的确寥寥可数。

历经数年的发展，Python 已今非昔比，各领域都不乏 Python 的成功案例。就拿 Web 方面来说，正如 PHP 给 Yahoo!带来的巨大动力，Python 在新一代的互联网霸主——Google 内部早就充当了重要角色，成为排名第三的“官方语言”。而就在前一段时间，Google 革命性的 App Engine 产品一经推出立即引起了莫大关注，其首选开发语言就是 Python。

纵观国内技术环境，Python 语言仍处于慢热的状态，应用仍然不算广泛。不过我们也已经有称得上比较成功的实现案例了，比如著名的 Web 2.0 的代表站点——豆瓣网即是用 Python 开发的，创始人杨勃对 Python 的效率和优雅赞誉有加。

Python 也是权威机构 TIOBE 评出的 2007 年度编程语言，这些“利好”的消息也将进而带动新一轮的技术走向，预示着 Python 更大规模的流行时代即将到来。

话说回来，“开放平台”在未来几年一定是个不可避免的技术趋势，而跟着大厂商的平台亦步亦趋，照猫画虎，想必也能开发出来繁多的周边应用，但开放未必对所有人都是个好事情，久而久之开发者难免有盲人摸象之感，很难掌握全局，掌握关键架构技术，故深入研究 Python 的基础技术仍不可少。这本《Python 源码剖析》的出版恰是好时机，弥补了国内图书在这方面的空白，此外，作者在 Python 领域的精耕细作的研究精神亦值得学习。

研读、分析源代码乃是提高编程技能的一条捷径，庖丁解牛方能游刃有余，夯实基础方可构建高楼大厦。

读这本《Python 源码剖析》就像一次探险之旅，祝愿朋友们能够获得一次愉悦的阅读体验。

冯大辉

2008 年 4 月于杭州

# 前言

## 缘起

第一次接触 Python，是通过《程序员》杂志上“恶魔吹着笛子来”的系列文章——《自由与繁荣的国度》。但是真正开始使用 Python，还是在进入实验室，开始研究自然语言处理和信息检索之后。自然语言处理其实大部分的时间都在与文本打交道，需要进行大量的对文本分析、统计的工作。开始的时候，我使用的是 C++，因为大学的时候第一门编程语言课就是 C，其后转向 C++是很自然的迁移。那时候觉得 C++很有一种高贵的感觉，因为 C++足够复杂，有足够多的 trick，尤其是像模板和泛型编程这样的新鲜玩意儿。掌握这么复杂的东西，也就意味着你的脑袋跟这东西一样复杂，这是很能让人虚荣的一件事。

C++的复杂性是个仁者见仁，智者见智的话题，但其实回到文本处理这个话题上来说，C++的 STL 在很大程度上已经足够好用了。文本处理不是服务器，所以不需要考虑自己管理内存，不需要考虑这个模式那个模式，STL 提供了足够多的工具，简单组装一下就可以用了。

但俗话说得好，“不怕不识货，就怕货比货”，当我开始尝试用 Python 来进行日常的工作之后，突然发现 C++太复杂了。对于 Python，我的感觉只有四个字：摧枯拉朽。我只需要简单地写一个 `l = [ ]`，再也不用写诸如 `list< map<string, string> > l = list< map<string, string> >()` 这样折磨人眼的东西了，这使得代码量急剧减少。对于采用 Python 这样的英明决策，我想，最满意的就是我的手指头了。

随着对 Python 的逐渐熟悉，我越来越惊叹于 Python 简洁的表达，强大的功能。尤其是 Python 表现出来的强烈的动态性。比如下面这段与解释器的交互过程：

```
>>> class A:
...     pass
...
>>> a = A()
>>> a.show()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: A instance has no attribute 'show'
>>>
>>> class B:
...     def show(self):
...             print 'i am B'
...
>>> A.__bases__ = (B,)
>>> a.show()
i am B
>>>
```

这样的动态能力，在当时简直让我目瞪口呆。从那时候起，我就有了一个强烈的好奇心：Python 是怎么实现的？我们知道 Python 是用 C 语言实现的，那这些神奇的动态能力是怎么通过 C 语言完成的呢？

于是我开始上网搜索资料，然而我发现，详细介绍动态语言实现原理的资料根本就没有，只有一些零散的信息散落在各种资料中。再具体到 Python，唯一一篇与 Python 实现相关的资料是《The Architecture of Python》，这是美国一所大学的学生在一次课程设计中产生的文档。这份文档篇幅太短，内容也太简略了，只包含了一些最简单的信息，即使对作为 Python 中对象模型关键的 PyObject 结构体，也仅仅有一些简单的描述。最要命的是，其研究的对象是 Python 2.1，版本太久远了。而我准备开始研究 Python 如何实现时，Python 已经进化到了 2.4，Python 的对象模型已经发生了重大的改变，所以这份文档对于想要深入掌握 Python 的实现来说，几乎没有太大作用，让人有一种“食之无味、弃之可惜”的感觉（这份文档目前在 Google 上已经搜索不到了）。但不管怎么说，这份文档给了我对于 Python 实现的一个最初的认识，给了我一个起始点。

2004 年年末，我开始了探索 Python 如何实现的漫漫长征。我选择了编译这个最初的切入点，但是很快我就发现，Python 的编译过程中大量地使用了 Python 中的一些内置对象，所以我将切入点转向了 Python 的对象模型。在完成了 Python 对象模型和内建对象的剖析后，又重新转回到编译过程的剖析，我发现 Python 的编译过程实际上就是一个标准的编译过程，在任何一本关于编译原理的书上，你都可以找到它的实现过程。于是，我做了一个决定，不再剖析 Python 的编译过程，而是以 Python 的编译结果为起点，开始 Python 虚拟机的剖析，正因为这样，你在本书中看不到对 Python 的编译过程的剖析，不过别着急，你能够看到 Python 的编译结果，对于理解 Python 虚拟机的实现来说，这个编译结果才是最重要的。

很快，第一篇关于 PyObject 的笔记出炉了，紧接着，Python 中最简单的对象——整数对象的笔记也诞生了。到了这个时候，我开始对完成这项工作有信心了，虽然后来的经历证明我当时的信心是多么的虚妄。也是在这个时候，我开始看到 Python 实现的精妙之处，完成这项工作的兴趣和动力也越来越强。

随着对 Python 挖掘的不断深入，我开始碰到一个又一个的瓶颈，虚拟机的框架、函数的实现、class 机制、module 的动态加载……在没有太多外部资料的情况下，要想通过源码弄明白一样东西，实在是太艰难了。有的时候，由于难度太大，我甚至中断了剖析的工作，直到很久以后，才在好奇心和兴趣的强烈驱使下重新开始。正是由于这样的磕磕绊绊，到了 2007 年底，这项工作才算完成。花了两年多的时间，除了难度方面的原因，另一个原因是 2006 年底，Python 社区发布了 Python 2.5，为了保证与最新的 Python 实现保持一致，我将所有的剖析跟 Python 2.5 又对照了一遍，所以，你手上这本书，跟最新的 Python 实现是一致的。

## 谁需要这本书

什么人需要这本书呢？呃，当然，我自然是希望需要这本书的人越多越好，我巴不得所有的程序员人手一本☺。从我自身的体会来说，我觉得以下三类人都会对这本书感兴趣：Python 程序员、动态语言爱好者、C 程序员。

Python 程序员当然能从这本书得到最多的收获，在这本书里，你能找到 Python 所有魔法的最终来源，每一个看似神奇的特性都会得到一个最朴素的解释，每一个所谓的高级特性，比如 decorator，都会变成你手中平凡的工具。当然，你还会对你编写的 Python 代码有一个最透彻的认识。比如下面这段代码，为什么不能加载 D 呢？

```
[A.py]
from B import D

class C:
  pass

[B.py]
from A import C

class D:
  pass
```

当最终完成对 Python 的剖析之后，我发现对于 Python 代码，可以有一种洞彻肺腑的感觉。看着代码，在脑海里就完全可以出现虚拟机在后台如何运作的情景，这种感觉非常棒。

除了 Python 程序员，其他的动态语言爱好者都能从这本书得到很多有益的信息。不论是 Python 在其他平台的实现——IronPython、Jython，还是其他的动态语言——Ruby、Javascript，只要你对动态语言感兴趣，想要了解动态语言的实现机理，那赶紧把这本书搬回家。动态语言在大致的架构和实现机制上，有很多的相似之处。

这本书在本质上是探讨如何用 C 语言实现 Python 这门动态语言的，所以 C 程序员能从中获得巨大的收益。作为一个优秀的开源项目，通过阅读它的源码，C 程序员可以从中汲取世界上最优秀的 C 程序员的经验。当然，对于另一部分特殊的 C 程序员，这本书的价值就更大了，如果你需要为 Python 编写 C 扩展模块，或者更酷地，你想要在你的系统中嵌入一个脚本引擎，就像很多网络游戏中的脚本引擎，那么本书就是你必备的了。

尽管这本书看上去讨论的是个高深的话题，但其实对读者知识储备的要求相当少。只需要 C 语言和 Python 语言的基础知识。

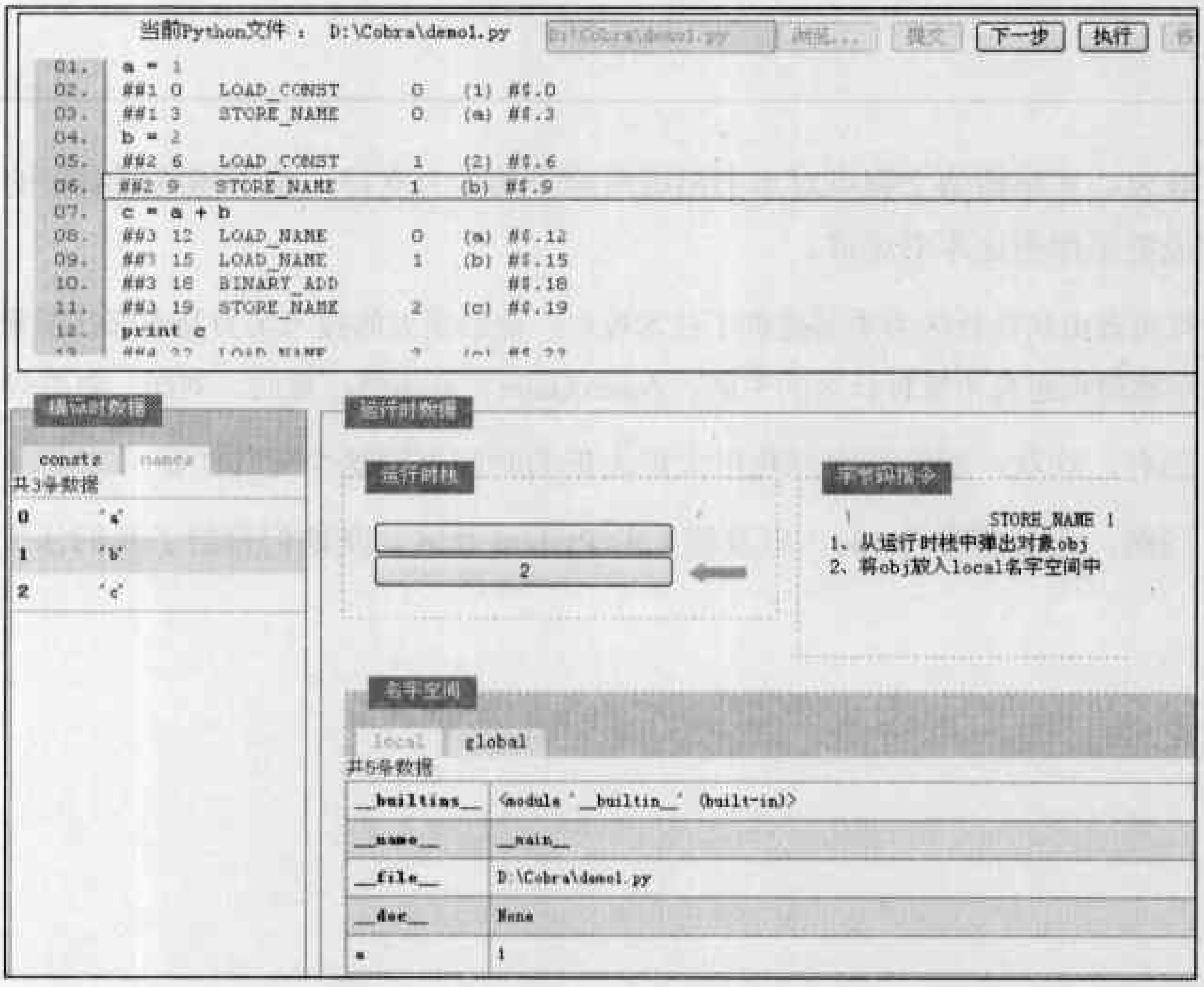
1. C 语言的基础知识是必须的，但是不必多么精通 C，只需要能看懂 C 代码就可以。

2. Python 语言的基础知识也是必须的，这更不是问题了，有编程基础的读者，半天的时间，就能看懂所有的 Python 语法，这本书基本上不涉及任何 Python 的库（在线程部分，有一些线程库的介绍），所以，你只需要熟悉基本的 Python 语法就可以了。

# 如何阅读这本书

关于这个问题，我的建议是，不要只阅读，更要实践。最基本的，你应该打开自己最熟悉的代码浏览工具，在真实的源码和书中的描述之间比对揣摩。更进一步，你可以随着本书的进展，动手捣鼓捣鼓 Python 的源码，用真实的输出验证书中的描述和你的理解。对于技术这个东西，再生花的妙笔也不能让你仅仅通过“阅读”便能乾坤在握，唯有亲身尝试，才能深解其中三昧。

为了帮助读者更好地利用此书，我在 Google Code 上发起了一个旨在可视化 Python 虚拟机的开源项目——Cobra（*http://code.google.com/p/python-cobra/*），其目的在于将 Python 虚拟机在执行一条条字节码指令时的运行时环境，以及虚拟机的状态变化，以可视的形式展现出来，以更加生动形象的方式加深读者对 Python 虚拟机的理解。同时，我也期望这个项目能成为有兴趣的读者锻炼自己改造 Python 虚拟机的能力的平台。该项目还处于发展初期，目前仅仅实现了一般表达式的可视化，我会尽快增强 Cobra 的功能，也希望感兴趣的读者一起加入到这个有趣的项目中来。下图是目前 Cobra 对简单的一般表达式的可视化效果。




# 代码下载

本书还提供了书中涉及到的两个简单项目的源代码，一个是对 Python 的最简化模拟的 Small Python，另一个是对 pyc 文件进行解析的解析器。两个项目的代码都可以在博文对本书的支持网站下载：*http://bv.csdn.net/resource/pythonympx.rar*。

# 联系作者

面对如此复杂而又伟大的 Python，以一本书的篇幅，难免对有的主题的涉及会显得简略；另一方面，以一人之力，在剖析与撰写中也难免会有错误与遗漏。如果你有什么建议，或者认为我还遗漏了什么东西，非常期待与你交流。你可以通过 *search.pythoner@gmail. com* 与我直接联系；或者，如果你愿意结识更多的 Python 同道，你也可以访问中文 Python 用户组（*https://groups.google.com/group/python-cn*），把你的心得与中国的 Python 爱好者一起分享，我也会一直在那里对本书进行支持。

## 致谢

我衷心地感谢博文视点对本书出版所给予的大力支持，由于杨绣国编辑的大力推动，我才能更早地把这本书完成。

哲思自由软件社区为本书提供了技术校对，他们强大的技术实力是本书高质量的保证，在此特别感谢哲思自由软件社区的王聪、Zoom.Quiet、夏清然、夏武、刘洋、董溥等。

还有，欢欢，感谢你容忍我用大把大把的时间来写这个劳什子☺

当然，还要感谢 Guido，以及伟大的 Python 社区，是他们带给了我们伟大的 Python。

# 联系博文视点

您可以通过如下方式与本书的出版方取得联系。

读者信箱：*reader@broadview.com.cn*

投稿信箱：*bvtougao@gmail.com*

北京博文视点资讯有限公司（武汉分部）

湖北省武汉市洪山区吴家湾邮科院路特 1 号湖北信息产业科技大厦 1402 室

邮政编码：430074

电　　话：（027）87690813

传　　真：（027）87690595

若您希望参加博文视点的有奖读者调查，或对写作和翻译感兴趣，欢迎您访问：

*http://bv.csdn.net*

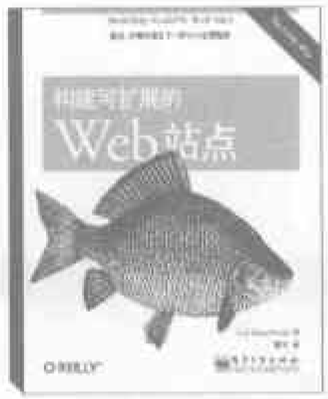
关于本书的勘误、资源下载及博文视点的最新书讯，欢迎您访问博文视点官方博客：

*http://blog.csdn.net/bvbook*

## 博文视点Web开发系列




《Python网络编程基础》
John Goerzen　著
莫迟　等译
■ Python网络编程最佳入门图书
■ FTP、Email、XML、Web Service、多线程、异步通信……
完整涵盖网络编程的方方面面




《构建可扩展的Web站点》
Cal Henderson　著
徐宁　译
■ Web 2.0代表网站Flickr总架构师
Cal Henderson经典力作



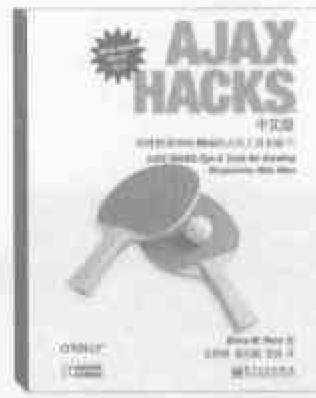

《Web开发人员参考大全》
Lázaro Issi Cohen,Joseph Issi Cohen　著
胡为君　译
■ 没有你查不到的属性
■ 没有你找不到的标签
■ 没有你用不到的知识



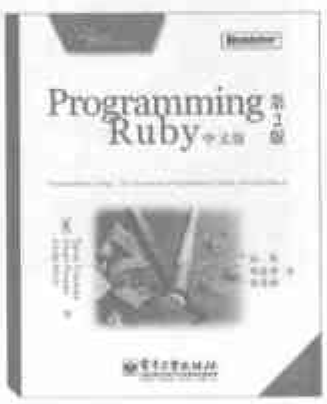

《Ajax模式与最佳实践》
Christian Gross　著
李锟　等译
■ 深邃洞察Web开发整体架构
■ 全面涵盖客户端与服务器端开发




《Ajax设计模式》
Michael Mahemoff　著
杨仁和　译　　Ajaxcn　审校
■ 全景展现Ajax技术多年智慧结晶
■ 全面总结Web 2.0开发技术架构




《AJAX HACKS中文版：创建快速响应Web站点的工具和技巧》
Bruce W. Perry　著
张险峰　等译
■ 80个AJAX独门绝技一网打尽




《Programming Ruby中文版，第2版》
Dave Thomas,Chad Fowler, Andy Hunt　著
孙勇　姚延栋　张海峰　译
■ 全球公认Ruby权威入门参考指南
■ 让编程成为你的直觉
■ 让开发成为你的乐趣

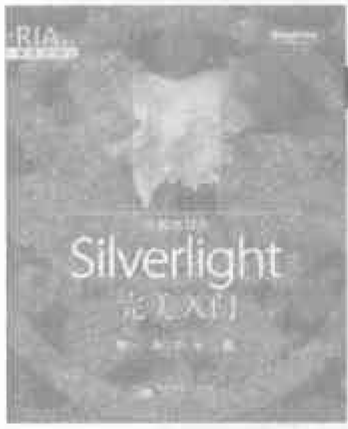



《Web开发敏捷之道
——应用Rails进行敏捷Web开发，第2版》
Dave Thomas, David Heinemeier Hansson　著
林芷薰　译　　透明　审校
■ 70%全新内容
■ Ruby On Rails 经典力作最新第2版
■ 第1版荣获Jolt Award（震撼大奖）




《PHP程序设计（第2版）》
Rasmus Lerdorf, Kevin Tatroe　等著
陈浩　胡丹　徐景　译
■ PHP 5的权威指南书籍
■ 包含PHP创始人Rasmus Lerdorf等PHP专家的独特见解
■ 中国四大权威PHP社区鼎力推荐

# 博文视点精品书廊系列




**《圣殿祭司的Silverlight完美入门》**

奚江华　著

- 完美展现下一代Web技术
- 圣殿祭司的独家精彩实例

**《Silverlight权威讲座——ASP.NET整合秘技与独家案例剖析》**

董大伟　著

- ASP.NET、ASP.NET Ajax、Silverlight真正完美结合
- 全面套件化，不会Ajax一样做出绚丽效果
- 独有精彩案例







**《SQL Server 2005 T-SQL数据库设计》**

胡百敬　姚巧玫　陈俊宇　杨先民　著

胡百敬　审校

- 台湾微软首席产品营销经理黄淑翠鼎力推荐
- SQL Server四大高手联袂打造
- 助你打通T-SQL学习的任督二脉

**《构建Oracle高可用环境——企业级高可用数据库架构、实战与经验总结》**

陈吉平　著

- 阿里巴巴首席DBA、DBA Manager、Oracle ACE 冯春培作序推荐
- 内容横跨主机、存储、数据库三大领域
- 全面介绍Oracle以及周边环境的高可用性







**《JAVASCRIPT语言精髓与编程实践》**

周爱民　著

- 《大道至简——软件工程实践者的思想》作者周爱民先生最新作品
- 详细讲述混合式语言JavaScript多种特性
- 深入探讨JavaScript的本质及其应用

## ……—全面升级AJAX素养




《ASP.NET AJAX应用剖析立即上手》
董大伟　著

■ ASP.NET AJAX Developer Day研讨会、MSDN讲座专属顶尖讲师、微软MVP董大伟倾力巨著




《Ajax and REST Recipes中文版》
Christian Gross　著
李琳骁　蔡毅　译

■ 解决您所有的Ajax /REST问题
■ 为您提供架构与开发问题的专家技巧




《Ajax修炼之道》
Justin Gehtland, Ben Galbraith, Dion Almaer　著
徐锋　胡冰　译

■ 各个层次的web开发人员都能获益匪浅
■ 来自Ajaxian.com权威专家的指导，解析Ajax框架的三层结构




《Ajax设计模式》
Michael Mahemoff　著
杨仁和　译　　Ajaxcn　审校

■ 全景展现Ajax技术多年智慧结晶
■ 全面总结Web2.0开发技术架构




《Ajax模式与最佳实践》
Christian Gross　著
李锟　张祖良　蔡毅　赵泽欣　译

■ 深邃洞察Web开发整体架构
■ 全面涵盖客户端与服务器端开发




《AJAX HACS中文版：创建快速响应Web站点的工具和技巧》
Bruce W.Perry　著
张险峰　张伟超　笑风　译

■ 80个Ajax独门绝技一网打尽

博文视点资讯有限公司（BROADVIEW Information Co.,Ltd.）是信息产业部直属的中央一级科技与教育出版社——电子工业出版社（PHEI）与国内最大的IT技术网站CSDN.NET和最具专业水准的IT杂志社《程序员》合资成立的以IT图书出版为主业、开展相关信息和知识增值服务的资讯公司。

我们的理念是：创新专业出版体制；培养职业出版队伍；打造精品出版品牌；完善全面出版服务。

秉承博文视点的理念，博文视点的产品线为面向IT专业人员的出版物和相关服务。博文视点将重点做好以下工作：

（1）在技术领域开发专业作（译）者群体和高质量的原创图书

（2）在图书领域建立专业的选题策划和审读机制

（3）在市场领域开创有效的宣传手段和营销渠道

博文视点有效地综合了电子工业出版社、《程序员》杂志社和CSDN.NET的资源和人才，建立全新专业的立体出版机制，确立独特的出版特色和优势，将打造IT出版领域的著名品牌，并力争成为中国最具影响力的专业IT出版和服务提供商。

作为合资公司，博文视点的团队融合了各方面的精英力量：原电子工业出版社IT图书专业出版实力的代表部门——计算机图书事业部的团队；《程序员》杂志社和CSDN网站的主创人员；著名IT专业图书策划人周筠女士及其创作群。这是一个整合专业技术人员和专业出版人员的团队；这是一个充满创新意识和创作激情的团队；这是一个不断进取、追求卓越的团队。

电子工业出版社与《程序员》杂志和CSDN网站的合作以最有效率的方式形成了出版资源、媒体资源、网络资源的整合和互动，成为2003年IT出版界备受瞩目的事件。

“技术凝聚实力，专业创新出版”，BROADVIEW与您携手共迎信息时代的机遇与挑战！

博文视点

地址：北京市万寿路173信箱电子工业出版社博文视点资讯有限公司　邮　编：100036

总　机：010-88254356　传　真：010-88254356-802

作译者服务热线(Fax)：027-66012955

读者服务热线：027-66012959　读者信箱：reader@broadview.com.cn

武汉分部地址：武汉市洪山区吴家湾湖北信息产业科技大厦1402室　邮编：430074

欢迎访问以下资源网站：

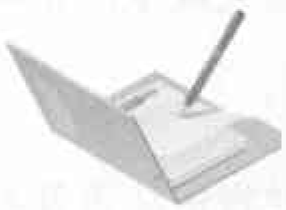
博文官方博客：http://blog.csdn.net/bvbook/

博文官方网站：http://www.broadview.com.cn/

# 在线有奖读者调查表

Broadview®
www.broadview.com.cn

《Python源码剖析
——深度探索动态语言核心技术》



http://bv.csdn.net

登录以上网站告诉我们您关于这本书的建议、意见

就有机会获赠博文视点的『**新书一本**』

并参加年终大抽奖活动

您的支持就是我们创造精品动力的源泉！

**欢迎投稿：** bvtougao@gmail.com

**读者信箱：** reader@broadview.com.cn

博文视点更多资源网站：

- VSTS虚拟社区：
  http://yishan.cc/
- 《代码大全》资源网站：
  http://www.cc2e.com.cn/
- 博文视点官方博客：
  http://blog.csdn.net/bvbook/

# 反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为，歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传　　真：（010）88254397

E-mail：　dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

　　　　　电子工业出版社总编办公室

邮　　编：100036

# 目录

第 0 章 Python 源码剖析——编译 Python ........ 1
0.1 Python 总体架构 ........ 1
0.2 Python 源代码的组织 ........ 2
0.3 Windows 环境下编译 Python ........ 4
0.4 Unix/Linux 环境下编译 Python ........ 7
0.5 修改 Python 源代码 ........ 7
0.6 通往 Python 之路 ........ 9
0.7 一些注意事项 ........ 10

第 1 部分 Python 内建对象 ........ 13

第 1 章 Python 对象初探 ........ 15
1.1 Python 内的对象 ........ 16
1.1.1 对象机制的基石——PyObject ........ 17
1.1.2 定长对象和变长对象 ........ 18
1.2 类型对象 ........ 19
1.2.1 对象的创建 ........ 20
1.2.2 对象的行为 ........ 22
1.2.3 类型的类型 ........ 24
1.3 Python 对象的多态性 ........ 25
1.4 引用计数 ........ 26
1.5 Python 对象的分类 ........ 28

第 2 章 Python 中的整数对象 ........ 29
2.1 初识 PyIntObject 对象 ........ 29
2.2 PyIntObject 对象的创建和维护 ........ 34

2.2.1 对象创建的 3 种途径 ......34
2.2.2 小整数对象 ......35
2.2.3 大整数对象 ......36
2.2.4 添加和删除 ......37
2.2.5 小整数对象池的初始化 ......43
2.3 Hack PyIntObject ......44

第 3 章 Python 中的字符串对象 ......47
3.1 PyStringObject 与 PyString_Type ......47
3.2 创建 PyStringObject 对象 ......49
3.3 字符串对象的 intern 机制 ......52
3.4 字符缓冲池 ......56
3.5 PyStringObject 效率相关问题 ......58
3.6 Hack PyStringObject ......60

第 4 章 Python 中的 List 对象 ......63
4.1 PyListObject 对象 ......63
4.2 PyListObject 对象的创建与维护 ......64
4.2.1 创建对象 ......64
4.2.2 设置元素 ......66
4.2.3 插入元素 ......68
4.2.4 删除元素 ......72
4.3 PyListObject 对象缓冲池 ......74
4.4 Hack PyListObject ......75

第 5 章 Python 中的 Dict 对象 ......77
5.1 散列表概述 ......78
5.2 PyDictObject ......79
5.2.1 关联容器的 entry ......79
5.2.2 关联容器的实现 ......80
5.3 PyDictObject 的创建和维护 ......82
5.3.1 PyDictObject 对象创建 ......82
5.3.2 PyDictObject 中的元素搜索 ......83
5.3.3 插入与删除 ......89
5.3.4 操作示例 ......95
5.4 PyDictObject 对象缓冲池 ......96
5.5 Hack PyDictObject ......97

第 6 章 最简单的 Python 模拟——Small Python ........ 101
6.1 Small Python ........ 101
6.2 对象机制 ........ 102
6.3 解释过程 ........ 106
6.4 交互式环境 ........ 108

第 2 部分 Python 虚拟机 ........ 111

第 7 章 Python 的编译结果——Code 对象与 pyc 文件 ........ 113
7.1 Python 程序的执行过程 ........ 113
7.2 Python 编译器的编译结果——PyCodeObject 对象 ........ 115
7.2.1 PyCodeObject 对象与 pyc 文件 ........ 115
7.2.2 Python 源码中的 PyCodeObject ........ 116
7.2.3 pyc 文件 ........ 117
7.2.4 在 Python 中访问 PyCodeObject 对象 ........ 120
7.3 Pyc 文件的生成 ........ 120
7.3.1 创建 pyc 文件的具体过程 ........ 120
7.3.2 向 pyc 文件写入字符串 ........ 124
7.3.3 一个 PyCodeObject，多个 PyCodeObject ........ 128
7.4 Python 的字节码 ........ 129
7.5 解析 pyc 文件 ........ 130

第 8 章 Python 虚拟机框架 ........ 133
8.1 Python 虚拟机中的执行环境 ........ 133
8.1.1 Python 源码中的 PyFrameObject ........ 136
8.1.2 PyFrameObject 中的动态内存空间 ........ 138
8.1.3 在 Python 中访问 PyFrameObject 对象 ........ 139
8.2 名字、作用域和名字空间 ........ 140
8.2.1 Python 程序的基础结构——module ........ 140
8.2.2 约束与名字空间 ........ 141
8.2.3 作用域与名字空间 ........ 142
8.3 Python 虚拟机的运行框架 ........ 149
8.4 Python 运行时环境初探 ........ 152

第 9 章 Python 虚拟机中的一般表达式 ........ 157
9.1 简单内建对象的创建 ........ 157

9.2 复杂内建对象的创建......163
9.3 其他一般表达式......166
9.3.1 符号搜索......166
9.3.2 数值运算......169
9.3.3 信息输出......171
第 10 章 Python 虚拟机中的控制流......173
10.1 Python 虚拟机中的 if 控制流......173
10.1.1 研究对象——if_control.py......173
10.1.2 比较操作......175
10.1.3 指令跳跃......179
10.2 Python 虚拟机中的 for 循环控制流......183
10.2.1 研究对象——for_control.py......184
10.2.2 循环控制结构的初始化......184
10.2.3 迭代控制......188
10.2.4 终止迭代......191
10.3 Python 虚拟机中的 while 循环控制结构......192
10.3.1 研究对象——while_control.py......192
10.3.2 循环终止......194
10.3.3 循环的正常运转......195
10.3.4 循环流程改变指令之 continue......195
10.3.5 循环流程改变指令之 break......196
10.4 Python 虚拟机中的异常控制流......197
10.4.1 Python 中的异常机制......197
10.4.2 Python 中的异常控制语义结构......207
第 11 章 Python 虚拟机中的函数机制......215
11.1 PyFunctionObject 对象......215
11.2 无参函数调用......217
11.2.1 函数对象的创建......217
11.2.2 函数调用......220
11.3 函数执行时的名字空间......223
11.4 函数参数的实现......225
11.4.1 参数类别......226
11.4.2 位置参数的传递......229
11.4.3 位置参数的访问......233

11.4.4 位置参数的默认值......235
11.4.5 扩展位置参数和扩展键参数......243
11.5 函数中局部变量的访问......246
11.6 嵌套函数、闭包与 decorator......248
11.6.1 实现闭包的基石......249
11.6.2 闭包的实现......251
11.6.3 Decorator......257
第 12 章 Python 虚拟机中的类机制......259
12.1 Python 中的对象模型......259
12.1.1 对象间的关系......260
12.1.2 <type 'type'>和<type 'object'>......262
12.2 从 type 对象到 class 对象......263
12.2.1 处理基类和 type 信息......266
12.2.2 处理基类列表......267
12.2.3 填充 tp_dict......268
12.3 用户自定义 class......282
12.3.1 创建 class 对象......283
12.4 从 class 对象到 instance 对象......293
12.5 访问 instance 对象中的属性......296
12.5.1 instance 对象中的__dict__......298
12.5.2 再论 descriptor......299
12.5.3 函数变身......302
12.5.4 无参函数的调用......304
12.5.5 带参函数的调用......307
12.5.6 Bound Method 和 Unbound Method......308
12.6 千变万化的 descriptor......310
第 3 部分 Python 高级话题......313
第 13 章 Python 运行环境初始化......315
13.1 线程环境初始化......315
13.1.1 线程模型回顾......315
13.1.2 初始化线程环境......316
13.2 系统 module 初始化......320
13.2.1 创建__builtin__ module......320

13.2.2 创建 sys module ........ 327
13.2.3 创建__main__ module ........ 330
13.2.4 设置 site-specific 的 module 的搜索路径 ........ 331
13.3 激活 Python 虚拟机 ........ 334
13.3.1 交互式运行方式 ........ 335
13.3.2 脚本文件运行方式 ........ 336
13.3.3 启动虚拟机 ........ 337
13.3.4 名字空间 ........ 339
第 14 章 Python 模块的动态加载机制 ........ 343
14.1 import 前奏曲 ........ 343
14.2 Python 中 import 机制的黑盒探测 ........ 347
14.2.1 标准 import ........ 347
14.2.2 嵌套 import ........ 351
14.2.3 import package ........ 352
14.2.4 from 与 import ........ 356
14.2.5 符号重命名 ........ 358
14.2.6 符号的销毁与重载 ........ 359
14.3 import 机制的实现 ........ 362
14.3.1 解析 module/package 树状结构 ........ 365
14.3.2 加载 module/pakcage ........ 370
14.3.3 from 与 import ........ 381
14.4 Python 中的 import 操作 ........ 382
14.4.1 import module ........ 383
14.4.2 import package ........ 383
14.4.3 from & import ........ 383
14.4.4 import & as ........ 384
14.4.5 reload ........ 385
14.4.6 内建 module：imp ........ 386
14.5 与 module 有关的名字空间问题 ........ 386
第 15 章 Python 多线程机制 ........ 391
15.1 GIL 与线程调度 ........ 391
15.2 初见 Python Thread ........ 394
15.3 Python 线程的创建 ........ 396
15.3.1 建立多线程环境 ........ 397

15.3.2 创建线程......402
15.4 Python 线程的调度......413
15.4.1 标准调度......413
15.4.2 阻塞调度......416
15.5 Python 子线程的销毁......419
15.6 Python 线程的用户级互斥与同步......420
15.6.1 用户级互斥与同步......420
15.6.2 Lock 对象......421
15.7 高级线程库——threading......423
15.7.1 Threading Module 概述......424
15.7.2 Threading 的线程同步工具......425
15.7.3 Threading 中的 Thread......426

第 16 章 Python 的内存管理机制......429
16.1 内存管理架构......429
16.2 小块空间的内存池......432
16.2.1 Block......432
16.2.2 Pool......434
16.2.3 arena......438
16.2.4 内存池......442
16.3 循环引用的垃圾收集......457
16.3.1 引用计数与垃圾收集......457
16.3.2 三色标记模型......458
16.4 Python 中的垃圾收集......459
16.4.1 可收集对象链表......460
16.4.2 分代的垃圾收集......463
16.4.3 Python 中的标记—清除方法......466
16.4.4 垃圾收集全景......475
16.4.5 Python 中的 gc 模块......477
16.4.6 总结......479

CHAPTER

第 0 章

准备工作

# Python 源码剖析——编译 Python

在开始分析 Python 的实现之前，我们有许多的准备工作要做。比如，首先应该了解一下 Python 的整体架构，以期对 Python 的实现有一个宏观的认识。此外，我们还要介绍如何从源代码编译出 Python 可执行程序。因为在整个剖析源码的过程中，一个最好的学习方法就是不断根据掌握的知识修改 Python 的源代码，以印证自己的猜想和知识。因此，本章的目的是为进入 Python 源码剖析做一个充足的准备。

## 0.1 Python 总体架构

在最高的层次上，Python 的整体架构可以分为三个主要的部分，整个架构如图 0-1 所示。在图的左边，是 Python 提供的大量的模块、库以及用户自定义的模块。比如在执行 `import os` 时，这个 os 就是 Python 内建的模块，当然用户还可以通过自定义模块来扩展 Python 系统。

在图 0-1 的右边，是 Python 的运行时环境，包括对象/类型系统（Object/Type structures）、内存分配器（Memory Allocator）和运行时状态信息（Current State of Python）。运行时状态维护了解释器在执行字节码时不同的状态（比如正常状态和异常状态）之间切换的动作，我们可以将它视为一个巨大而复杂的有穷状态机。内存分配器则全权负责 Python 中创建对象时，对内存的申请工作，实际上它就是 Python 运行时与 C 中 `malloc` 的一层接口。而对象/类型系统则包含了 Python 中存在的各种内建对象，比如整数、`list` 和 `dict`，以及各种用户自定义的类型和对象。

在中间的部分，可以看到 Python 的核心——解释器（interpreter），或者称为虚拟机。在解析器中，箭头的方向指示了 Python 运行过程中的数据流方向。其中 Scanner 对应词法分析，将文件输入的 Python 源代码或从命令行输入的一行行 Python 代码切分为一个的 token；Parser 对应语法分析，在 Scanner 的分析结果上进行语法分析，建立抽象语法树（AST）；Compiler 是根据建立的 AST 生成指令集合——Python 字节码（byte code），就像 Java 编译器和 C#编译器所做的那样；最后由 Code Evaluator 来执行这些字节码。因此，Code Evaluator 又可以被称为虚拟机。

图中，在解释器与右边的对象/类型系统、内存分配器之间的箭头表示“使用”关系；而与运行时状态之间的箭头表示“修改”关系，即 Python 在执行的过程中会不断地修改当前解释器所处的状态，在不同的状态之间切换。

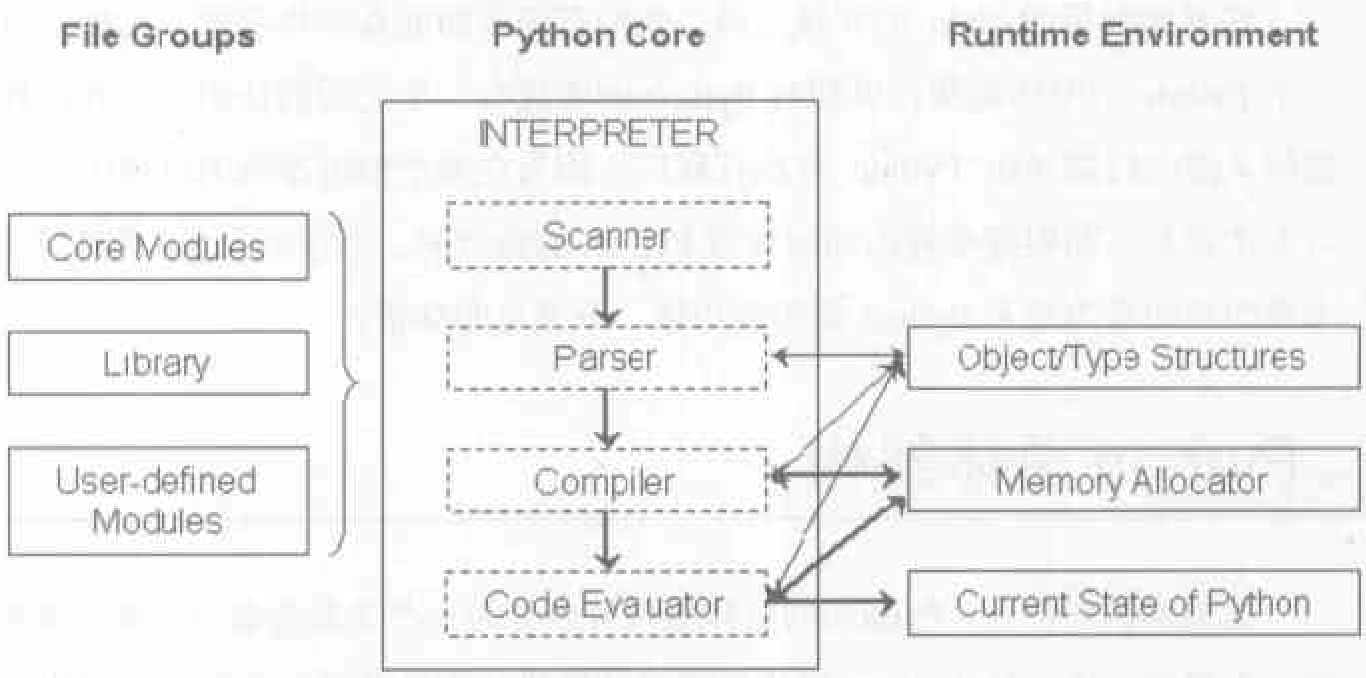



图 0-1 Python 总体架构

## 0.2 Python 源代码的组织

中国有句老话——巧妇难为无米之炊。要分析 Python 源码，首先要获得 Python 源码。Python 源码可以从 Python 的官方网站（*http://www.python.org*）（如图 0-2 所示）自由下载。当前 Python 的最新发布版本是 2.5.2，在本书中，我们剖析的对象是 2006 年 12 月 19 日正式发布的 Python 2.5。




图 0-2 下载 Python2.5 源码

下载了 Python 的源代码压缩包并解压后，可以看到如图 0-3 所示的目录结构。下面列出了一些主要目录包含的文件：

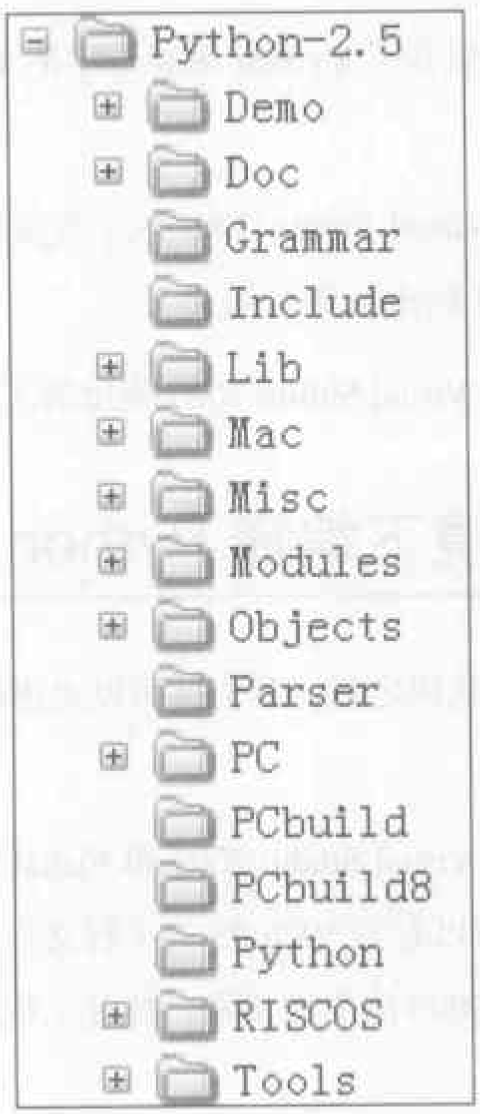



图 0-3 Python 目录结构

Include ：该目录下包含了 Python 提供的所有头文件，如果用户需要自己用 C 或 C++ 来编写自定义模块扩展 Python，那么就需要用到这里提供的头文件。

Lib ：该目录包含了 Python 自带的所有标准库，Lib 中的库都是用 Python 语言编写的。

Modules ：该目录中包含了所有用 C 语言编写的模块，比如 `random`、`cStringIO` 等。Modules 中的模块是那些对速度要求非常严格的模块，而有一些对速度没有太严格要求的模块，比如 `os`，就是用 Python 编写，并且放在 Lib 目录下的。

Parser ：该目录中包含了 Python 解释器中的 Scanner 和 Parser 部分，即对 Python 源代码进行词法分析和语法分析的部分。除了这些，Parser 目录下还包含了一些有用的工具，这些工具能够根据 Python 语言的语法自动生成 Python 语言的词法和语法分析器，与 YACC 非常类似。

Objects ：该目录中包含了所有 Python 的内建对象，包括整数、`list`、`dict` 等。同时，该目录还包括了 Python 在运行时需要的所有的内部使用对象的实现。

Python ：该目录下包含了 Python 解释器中的 Compiler 和执行引擎部分，是 Python 运行的核心所在。

PCBuild ：包含了 Visual Studio 2003 的工程文件，研究 Python 源代码就从这里开始（本书将采用 VS2003 对 Python 进行编译）。

PCBuild8 ：包含了 Visual Stuido 2005 使用的工程文件。

## 0.3 Windows 环境下编译 Python

下载了 Python 的源代码之后，我们就可以走出剖析 Python 源码的第一步了——编译 Python。

Python 2.5 提供了在 Visual Studio 2003 和 Visual Studio 2005 环境下进行开发的工程文件，在 PCBuild 目录下可以看到 VS2003 的工程文件，PCBuild8 目录下是 VS2005 的工程文件。这里使用的是 VS2003 的工程文件。打开工程文件后，我们还需要进行一些设置，才能成功编译。

首先，我们需要激活 VS2003 的配置对话框（如图 0-4 所示）：

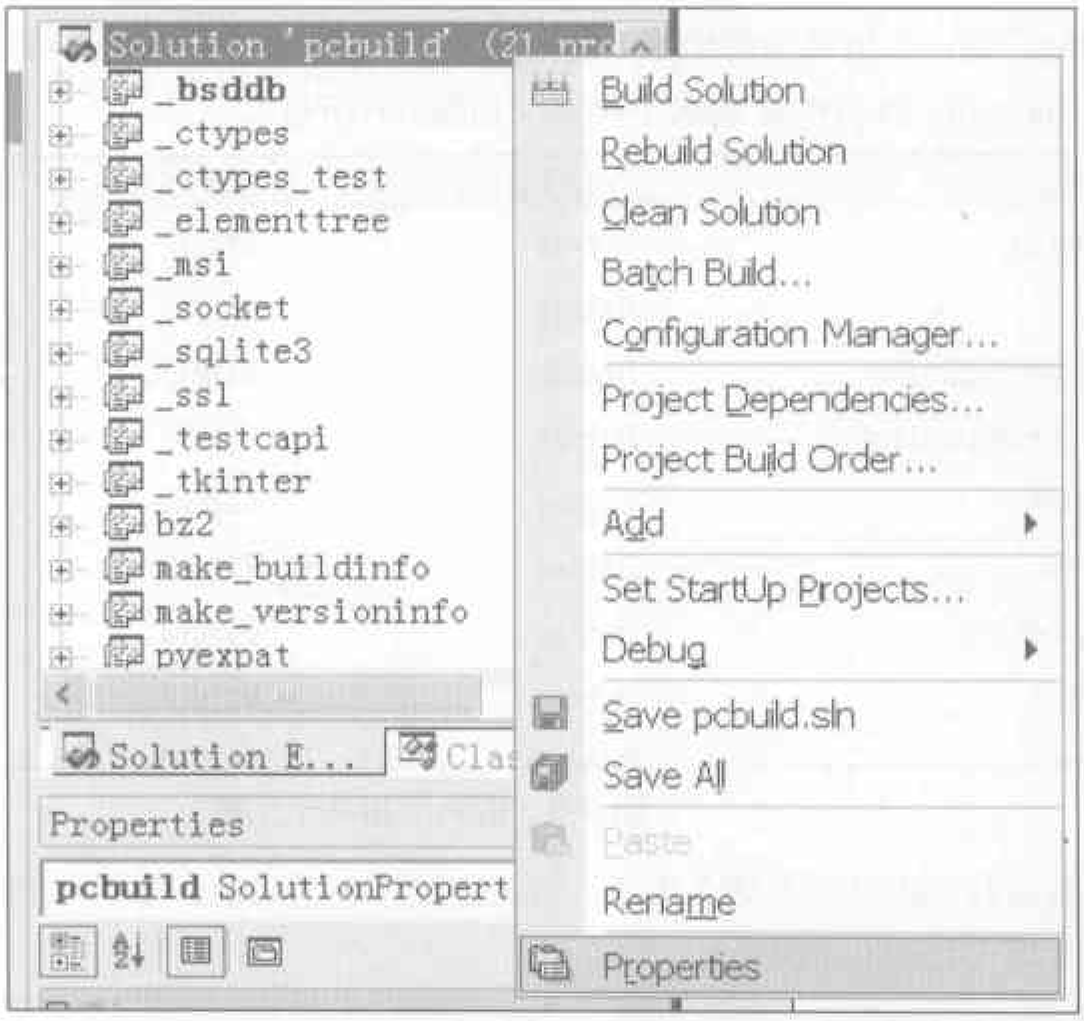



图 0-4 调出设置属性对话框

在配置对话框中，首先要做的就是更改 Startup Project，Python2.5 中默认设置的是 _bsddb，我们需要将其改为 Python（如图 0-5 所示）。

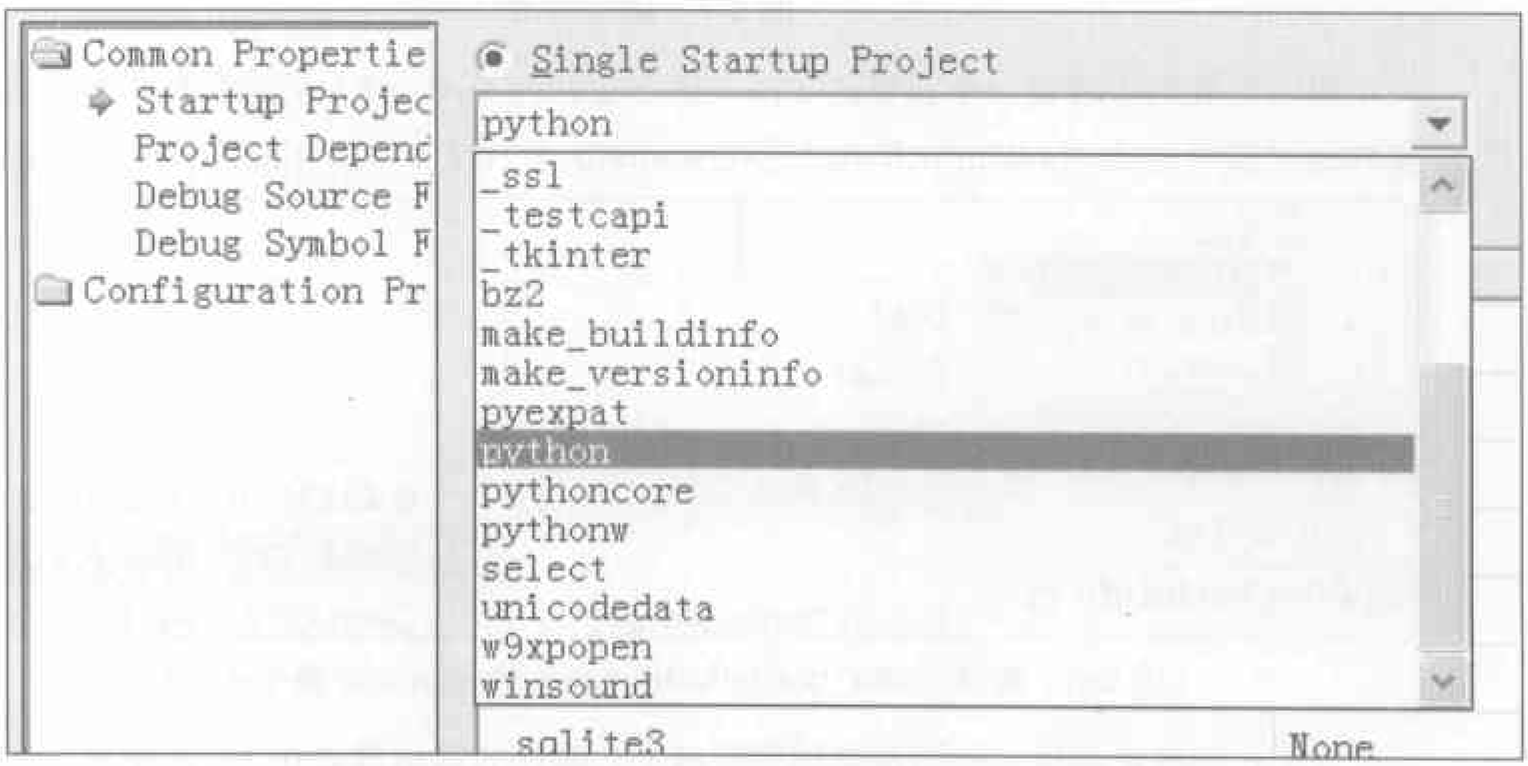



图 0-5 改变 startup project

由于我们剖析的只是 Python 的核心部分，不会涉及工程中的一些标准库和其他的模块，所以可以将它们从编译的列表中删除。点击配置对话框左边列表框中的“Configuration

Properties”后，会出现当前配置为需要编译的子工程，取消多余的子工程的选中状态，只保留 pythoncore 和 python 的选中状态（如图 0-6 所示）。

| Project | Configuration | Platform | Build |
|---|---|---|---|
| _tkinter | Debug | Win32 | ☐ |
| bz2 | Debug | Win32 | ☐ |
| make_buildinfo | Debug | Win32 | ☐ |
| make_versioninfo | Debug | Win32 | ☐ |
| pyexpat | Debug | Win32 | ☐ |
| python | Debug | Win32 | ☑ |
| pythoncore | Debug | Win32 | ☑ |
| pythonw | Debug | Win32 | ☐ |
| select | Debug | Win32 | ☐ |

图 0-6 取消不相关子工程

需要进行的改动就是这么多了，但是完成这些改动后，如果马上开始编译，那么编译还是会失败（如图 0-7 所示）：

| ! | ☑ | Description |
|---|---|---|
| | | Click here to add a new task |
| ! | ☐ | fatal error C1083: Cannot open precompiled header file: '.\x86-temp |
| ! | ☐ | fatal error LNK1181: cannot open input file '.\python25_d.lib' |

图 0-7 编译失败

原因是我们还需要一个必要的文件，这个文件在 Python2.5 的源码包中没有提供，必须要通过编译 make_buildinfo 和 make_versioninfo 子工程（如图 0-8 所示）才能生成：

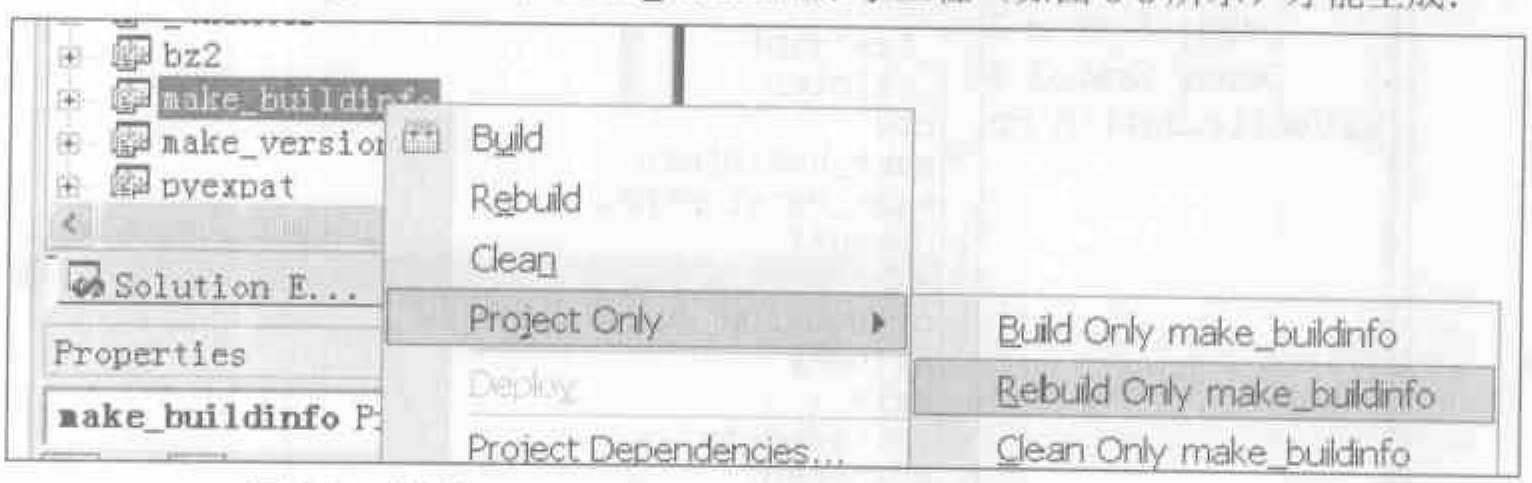



图 0-8 编译 make_buildinfo 和 make_versioninfo 两个子工程

好了，现在再编译，一切都会顺利完成了。编译的结果都放在 build 文件夹下，主要有两个：python25.dll 和 python.exe。可以看到 python.exe 非常小，实际上 Python 解释器的全部代码都在 python25.dll 中。对于 WinXP 操作系统，在安装 python 时，这个 python25.dll 会被拷贝到 C:\Windows\system32 下。

## 0.4 Unix/Linux 环境下编译 Python

在 Linux 环境下编译 Python 就没有那么麻烦了，按照标准的 tarball 安装软件的流程，顺序使用下列三个命令，即可完成 Python 的安装与编译：

- ./configure –prefix=<你期望 Python 安装到的目录的路径>
- make
- make install

经过这三个步骤，我们在第一步中指定的安装路径下就会显示 Python 安装的结果。目录 bin 下存放的是可执行文件；目录 lib 下存放的是 Python 的标准库；lib/python2.5/config 下存放的是 libpython2.5.a，用 C 语言对 Python 进行扩展时需要用到这个静态库。

需要注意的是，这样编译之后的结果，bin 目录下的 Python 可执行文件是静态链接的。如果想要编译成动态链接的，即编译的结果中出现 libpython2.5.so，那么需要在第一步加入“enable-shared”这个指令，从而编译后得到的 libpython2.5.so 就会出现在 lib 目录下。

以后我们在对 Python 源码进行修改时，都不会使用系统相关的 API，所以我们添加的代码都没有平台相关性问题，无论是 Windows 环境，还是 Unix/Linux 环境，都可以正常编译并运行。

## 0.5 修改 Python 源代码

修改源代码？这还有什么好说的吗？打开 VS2003（Windows 环境）或者 vi（Linux 环境），冲上去一顿敲打键盘不就 OK 了吗。确实，修改源代码其实没有什么好说的，不过有一些以后修改源代码时需要注意的事项，这里先提一下。

如果要观察 Python 执行过程中的动态的行为，当然可以使用 VS 自带的 debugger，不过有时候也需要在其中输出一些信息。对于输出信息，使用 `printf` 最简单。但是 `printf` 要输出 Python 中的某个对象却不是那么方便，幸好 Python 的 C API 中提供了一个输出对象的接口：

```
[object.h]
int PyObject_Print(PyObject *, FILE *, int);
```

比如我们在 `int_print` 中输出一个整数，可以将 `int_print` 修改成如下的函数：

```
[intobject.c]
static int int_print(PyIntObject *v, FILE *fp, int flags)
{
  //add by Robert
```

```
    PyObject* str = PyString_FromString("i am in int_print");
    PyObject_Print(str, stdout, 0);
    printf("\n");

    fprintf(fp, "%ld", v->ob_ival);
    return 0;
}
```

其中粗体部分为我们加入的代码，`PyString_FromString` 是 Python 提供的另一个 C API，用于从 C 中的原生字符数组创建出 Python 中的字符串对象。具体它是如何实现的，现在我们先不关心，只需要了解它的功能就 OK 了。

将 Python 重新编译后，得到新的 python25.dll，用这个 dll 去替换 system32 目录下的那个 python25.dll，我们就可以从 Python 运行时看到希望输出的结果如图 0-9 所示：

```
>>> print 100
'i am in int_print'
100
```

图 0-9 在 Python 源码中输出额外信息

在 `PyObject_Print` 中，第二个参数指明的是输出目标。上面的例子使用了 `stdout`，指定了输出目标为标准输出，当我们从命令行环境中激活 Python 时，没有问题，但是如果使用 IDLE 的话，就会发现，输出的信息没有了。原因是 IDLE 的输出目标已经不是 `stdout` 了，说明加入的输出代码失效了。

在 Python 中，有一个特性——可以自己重定向标准输出，考虑图 0-10 所示的例子：

```
>>> import sys
>>> sys.stdout
<open file '<stdout>', mode 'w' at 0x00B6F068>
>>> sys.stdout = open('my_stdout.txt', 'w')
>>> sys.stdout
>>>
```

图 0-10 重定向标准输出

开始时，标准输出是 `<stdout>`，这也是 C 中 `stdout` 所代表的那个系统的标准输出。随后，我们将 Python 的标准输出定向到了 my_stdout.txt 中。当再次输入“`sys.stdout`”时，我们发现屏幕上没有任何输出了，因为这时标准输出已经被重定向，所以现在输出的内容已经到 my_stdout.txt 中，在图 0-11 中显示了 my_stdout.txt 中的内容：

```
my_stdout.txt
0         10        20        30        40        50
1 <open file 'my_stdout.txt', mode 'w' at 0x00BD6DA0>
```

图 0-11 重定向后的标准输出——my_stdout.txt

所以，同样地，如果想让自己添加的代码输出到 IDLE 中，我们也必须使用重定向之后的标准输出，而不能再使用 stdout 这个系统标准输出了。在 Python 中，我们能通过 sys.out 访问到 Python 的标准输出，那么在 C 一级呢？Python 也提供了这样的 C API。下面的代码显示了如何输出到重定向后的标准输出：

```
static PyObject* int_repr(PyIntObject *v)
{
  ……
  //add by Robert
  If(PyInt_AsLong(v) == -999) {
     PyObject* str = PyString_FromString("i am in int_repr");
     PyObject* out = PySys_GetObject("stdout");
     if(out != NULL) {
         PyObject_Print(str, stdout, 0);
         printf("\n");
     }
  }
  ……
}
```

图 0-12 显示了在 IDLE 中输出的结果：

```
IDLE 1.2
>>> import sys
>>> sys.stdout
<idlelib.rpc.RPCProxy object at 0x00DA2F70>
>>> print -999
i am in int_repr
-999
>>>
```

图 0-12 在 IDLE 中输出信息

PyInt_AsLong 的功能是将 Python 的整数对象转换为 C 中的 int 值。这里我们还展示了在监视 Python 内部运行时的一个技巧——设置条件，这个技巧不论是在通过输出查看内部状态时还是使用 VS 的 debugger 查看内部状态时，都非常有用。Python 在启动时会进行初始化，在初始化时，会调用很多函数，其中可能包含我们想监视的函数，如果这个时候在其中不加任何条件地输出或是设置断点，我们会发现要么输出的信息将想监视的函数淹没了，要么 debugger 在断点处停留时和我们预想的情况完全不一样。

## 0.6 通往 Python 之路

在第 1 节的图 0-1 中，我们看到了 Python 在整体上的三巨头，但是在本书中，不会对所有的内容都进行剖析。本书的目标是彻底剖析 Python 在运行时的行为，从而为 Python 程序员彻底理解 Python 的运行机制打下坚实的基础。当然，如果你彻底理解了 Python 的

运行时行为，那么对于如何利用Python的C API来编写Python扩展，如何在C中内嵌Python解释器等，都变得水到渠成。

本书将剖析的重点放在 Python 解释器和 Python 运行时环境上，对于 Python 的大量的外部库，我们将在需要的时候进行剖析。

在 Python 解释器中，我们将大量的精力放在虚拟机这一部分。对于词法解析，语法解析和编译，本书将不完全涉及。因为对于 Python 运行时行为的理解，与虚拟机非常相关，而跟 Python 的编译过程没有太大的关系。不过，Python 编译的结果，编译所得的字节码指令，对于运行时行为则大有影响。所以书中会将 Python 编译结果和 Python 虚拟机结合起来，一起进行剖析。

本书将对 Python 源码的剖析分为下面三个部分：

- 第 1 部分：Python 内建对象。主要内容是简要介绍 Python 对象模型，以及剖析主要内建对象，包括整数、字符串、list 和 dict。在对内建对象的剖析中，我们会深入其实现，细致地分析各种对象在 C 一级是如何被构建起来的。
- 第 2 部分：Python 虚拟机。主要内容是分析 Python 虚拟机执行字节码指令的过程。在这一部分中，我们将看到 Python 是如何通过虚拟机实现各种表达式、控制流、异常机制、函数机制及类机制。
- 第 3 部分：Python 高级话题。主要内容是剖析 Python 的运行环境以及一些高级话题。内容包括：Python 运行环境的初始化、动态加载机制、多线程机制和内存管理机制。

## 0.7 一些注意事项

在正式进入对 Python 的剖析之前，有一些在阅读代码时需要注意的事项，这里先提一下：

➢ 在 Python 2.4 的源码中，许多数值的类型都是 int 或 long，而在 Python 2.5 的源码中，Python 自定义了一个新的类型 `Py_ssize_t`。一般的，凡出现这个类型的地方，都可以以 int 视之。对于非 Python 开发者阅读代码和理解 Python 来说，不需要在意其间的差别。

➢ Python有一套相当复杂的内存管理机制。同时，由于历史的原因，也有一套相当混乱的内存管理接口。如果在一开始就剖析其内存管理机制，我想就会令读者望而却步，所以我们把对内存管理机制的剖析放到了第 3 部分。但是，在第 1 部分中你就会看到对内存管理接口的使用。因为创建对象必先分配内存，而它必须通过内存管理接口，所以我们在这里需要对这些内存管理接口进行一下概念上的简化，简化成大家熟悉的

接口。通常 Python 的源码中会使用 PyObject_GC_New、PyObject_GC_Malloc、PyMem_MALLOC，PyObject_MALLOC 等 API。只要坚持一个原则，即凡是以 New 结尾的，都以 C++中的 new 操作符视之；凡是以 Malloc 结尾的，都以 C 中的 malloc 操作符视之。下面是几个例子：

```
[PyString_FromString() in stringobject.c]
op = (PyStringObject *)PyObject_MALLOC(sizeof(PyStringObject) + size);
等效于:
PyStringObject* op =
(PyStringObject*)malloc(sizeof(PyStringObject)+size);

[PyList_New() in listobject.c]
op = PyObject_GC_New(PyListObject, &PyList_Type);
等效于:
PyListObject* op = new PyList_Type();

op->ob_item = (PyObject **) PyMem_MALLOC(nbytes);
等效于:
op->ob_item = (PyObject **)malloc(nbytes);
```

- 在本书中，我使用了大量的图片，一图胜千言。在这些图片中，大量使用的构图元素就是箭头。在大多数情况下，箭头是用来表示 C 中的指针这个概念的，但是也有某些情况下箭头并不表示指针，这个通过上下文的信息可以判断出来；另一点需要注意的是，基于 C 中的指针是指向某一内存的起始地址，所以在图中，箭头指向的是内存块的边界，而不是内存块本身。由于边界本身就应该是两块内存的交界，所以这个箭头指向的是两块内存中的哪块内存就存在着模糊。在本书中，指针指向的内存块都是距离指针最近的向右或向下的那块内存，图 0-13 中给出了一个例子：



图 0-13 指针图示

在图 0-13 中，深色的方块就是指针所指向的内存块。

# 第 1 部分

# Python 内建对象

CHAPTER

第 1 章

# Python 对象初探

对象是 Python 中最核心的一个概念，在 Python 的世界中，一切都是对象，一个整数是一个对象，一个字符串也是一个对象。更为奇妙的是，类型也是一种对象，整数类型是一个对象，字符串类型也是一个对象。换句话说，面向对象理论中的“类”和“对象”这两个概念在 Python 中都是通过 Python 内的对象来实现的。

在 Python 中，已经预先定义了一些类型对象，比如 `int` 类型、`string` 类型、`dict` 类型等，这些我们称之为内建类型对象。这些类型对象实现了面向对象中“类”的概念；这些内建类型对象通过“实例化”，可以创建内建类型对象的实例对象，比如 `int` 对象、`string` 对象、`dict` 对象。类似的，这些实例对象可以视为面向对象理论中“对象”这个概念在 Python 中的体现。

同时，Python 还允许程序员通过 `class A(object)` 这样的表达式自己定义类型对象。基于这些类型对象，同样可以进行“实例化”的操作，创建的对象称为“实例对象”。Python 中不光有着这些千差万别的对象，这些对象之间还存在着各种复杂的关系，从而构成了我们称之为“类型系统”或“对象体系”的东西。

Python 中的对象体系是一个庞大而复杂的体系，如果说在本书的第一章我就试图将这个体系阐释清楚，这只能说明我是个疯子。在本章中，我们的重点将放在了解对象在 Python 内部是如何表示的，更确切地说，因为 Python 是由 C 实现的，所以我们首先要弄清楚的一个问题就是：对象，这个神奇的东西，在 C 的层面，究竟长得是个什么模样，究竟是三头六臂，还是烈焰红唇。

除此之外，我们还将了解到类型对象在 C 的层面是如何实现的，并初步认识类型对象的作用及它与实例对象的关系。总之，本章对 Python 对象体系的介绍力求简洁，但是并不肤浅，有的地方甚至会相当深入。因此，在本章的阅读中，如果有什么疑难的地方，没有关系，先放下，只要有一个直观的感觉就可以了，这并不妨碍你阅读接下来的内容。

本章的目的是为能够顺利而快速地进入对内建对象的剖析打下必要的基础，至于对 Python 对象体系的详细剖析，会在第 2 部分的最后一章中介绍到。只有到了那个时候，我们才有足够的能力将这个体系看个明白。

## 1.1 Python 内的对象

从 1989 年 Guido 在圣诞节揭开 Python 的大幕开始，一直到现在，Python 经历了一次一次的升级，但是其实现语言一直都是 ANSI C。我们知道，C 并不是一个面向对象的语言，那么在 Python 中，它的对象机制是如何实现的呢？

对于人的思维来说，对象是一个比较形象的概念，而对于计算机来说，对象却是一个抽象的概念。它并不能理解这是一个整数，那是一个字符串，对于计算机来说，它所知道的一切都是字节。通常的说法是，对象是数据以及基于这些数据的操作的集合。在计算机中，一个对象实际上就是一片被分配的内存空间，这些内存可能是连续的，也可能是离散的，这都不重要，重要的是这片内存在更高的层次上可以作为一个整体来考虑，这个整体就是一个对象。在这片内存中，存储着一系列的数据以及可以对这些数据进行修改或读取操作的一系列代码。

在 Python 中，对象就是为 C 中的结构体在堆上申请的一块内存，一般来说，对象是不能被静态初始化的，并且也不能在栈空间上生存。唯一的例外就是类型对象，Python 中所有的内建的类型对象（如整数类型对象，字符串类型对象）都是被静态初始化的。

在 Python 中，一个对象一旦被创建，它在内存中的大小就是不变的了。这就意味着那些需要容纳可变长度数据的对象只能在对象内维护一个指向一块可变大小的内存区域的指针。为什么要设定这样一条特殊的规则呢，因为遵循这样的规则可以使通过指针维护对象的工作变得非常的简单。一旦允许对象的大小可在运行期改变，我们就可以考虑如下的情形。在内存中有对象 A，并且其后紧跟着对象 B。如果运行期某个时刻，A 的大小增大了，这意味着必须将 A 整个移动到内存中的其他位置，否则 A 增大的部分将覆盖原本属于 B 的数据。只要将 A 移动到内存中的其他位置，那么所有指向 A 的指针就必须立即得到更新，光是想一想，就知道这样的工作是多么的繁琐。

## 1.1.1 对象机制的基石——PyObject

在 Python 中，所有的东西都是对象，而所有的对象都拥有一些相同的内容，这些内容在 PyObject 中定义，PyObject 是整个 Python 对象机制的核心。

```
[object.h]
typedef struct _object {
    PyObject_HEAD
} PyObject;
```

这个结构体是 Python 对象机制的核心基石，从代码中可以看到，Python 对象的秘密都隐藏在 PyObject_HEAD 这个宏中。

```
[object.h]
#ifdef Py_TRACE_REFS
/* Define pointers to support a doubly-linked list of all live heap objects. */
    #define _PyObject_HEAD_EXTRA            \
        struct _object *_ob_next;      \
        struct _object *_ob_prev;
    #define _PyObject_EXTRA_INIT 0, 0,
#else
    #define _PyObject_HEAD_EXTRA
    #define _PyObject_EXTRA_INIT
#endif

/* PyObject_HEAD defines the initial segment of every PyObject. */
#define PyObject_HEAD               \
    _PyObject_HEAD_EXTRA            \
    int ob_refcnt;              \
    struct _typeobject *ob_type;
```

当我们在 Visual Studio 的 release 模式下编译 Python 时，是不会定义符号 Py_TRACE_REFS 的。所以在实际发布的 Python 中，PyObject 的定义非常简单：

```
[object.h]
typedef struct _object {
    int ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

在 PyObject 的定义中，整型变量 ob_refcnt 与 Python 的内存管理机制有关，它实现了基于引用计数的垃圾收集机制。对于某一个对象 A，当有一个新的 PyObject *引用该对象时，A 的引用计数应该增加；而当这个 PyObject *被删除时，A 的引用计数应该减少。当 A 的引用计数减少到 0 时，A 就可以从堆上被删除，以释放出内存供别的对象使用。

在 ob_refcnt 之外，我们注意到 ob_type 是一个指向_typeobject 结构体的指针，那么这个结构体是一个什么东西呢？实际上这个结构体对应着 Python 内部的一种特殊对象，它是用来指定一个对象类型的类型对象。这个类型对象我们将在后边详细地分析。现

在我们看到了，在 Python 中，对象机制的核心其实非常简单，一个是引用计数，一个就是类型信息。

在 PyObject 中定义了每一个 Python 对象都必须有的内容，这些内容将出现在每一个 Python 对象所占有的内存的最开始的字节中。这句话的另一个意思是，每一个 Python 对象除了必须有这个 PyObject 内容外，似乎还应该占有一些额外的内存，放置些其他的东西。没错，倘若所有的 Python 对象都只包含 PyObject，那 Python 中岂不是只有唯一的一种对象了，这可是大大的不妙。在 PyObject 中定义的内容仅仅是每一个 Python 对象都必须拥有的一部分内容，以我们将在下一章剖析的整数对象为例子，你可以看到对象中除 PyObject 之外“其他的东西”究竟是些什么东西。

```
[intobject.h]
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

Python 的整数对象中，除了 PyObject，还有一个额外的 long 变量，我们说一个整数当然应该有一个值，这个“值”的信息就保存在 ob_ival 中。同样，Python 中的字符串对象、list 对象、dict 对象、成千上万的其他对象，都在 PyObject 之外保存了属于自己的特殊的信息。

## 1.1.2 定长对象和变长对象

整数对象的特殊信息是一个 C 中的整形变量，无论这个整数对象的值有多大，都可以保存在这个整形变量(ob_ival)中。但是很不幸，对于另一类对象，就没这么幸运了。如果你是 Python 的设计师，考虑一下应该如何实现字符串对象。很显然，类似于 PyIntObject，在 PyObject 之外，字符串对象应该维护“一个字符串”，但在 C 中，没有“一个字符串”这样的概念，所以准确的说法是，字符串对象应该维护“n 个 char 型变量”。这种对象实际上不光是字符串对象，比如对应于 C++中 list 或 vector 的列表对象，它也应该维护“n 个 PyObject 对象”。看上去这种“n 个……”似乎也是一类 Python 对象的共同特征，因此，Python 在 PyObject 对象之外，还有一个表示这类对象的结构体——PyVarObject：

```
[object.h]
#define PyObject_VAR_HEAD        \
    PyObject_HEAD           \
    int ob_size; /* Number of items in variable part */

typedef struct {
    PyObject_VAR_HEAD
} PyVarObject;
```

我们把整数对象这样不包含可变长度数据的对象称为“定长对象”，而字符串对象这

样包含可变长度数据的对象称为“变长对象”，它们的区别在于定长对象的不同对象占用的内存大小是一样的，而变长对象的不同对象占用的内存可能是不一样的。比如，整数对象“1”和“100”占用的内存大小都是 `sizeof(PyIntObject)`，而字符串对象“Python”和“Ruby”占用的内存大小就不同了。正是这种区别导致了 `PyVarObject` 对象中 `ob_size` 的出现。变长对象通常都是容器，`ob_size` 这个成员实际上就是指明了变长对象中一共容纳了多少个元素。注意，`ob_size` 指明的是所容纳**元素的个数**，而不是字节的数量。比如对于 Python 中最常用的 `list`，它就是一个 `PyVarObject` 对象，如果某一时刻，这个 `list` 中有 5 个元素，那么 `ob_size` 的值就是 5。

从 `PyObject_VAR_HEAD` 的定义可以看出，`PyVarObject` 实际上只是对 `PyObject` 的一个扩展而已。因此，对于任何一个 `PyVarObject`，其所占用的内存，开始部分的字节的意义和 `PyObject` 是一样的。换句话说，在 Python 内部，每一个对象都拥有相同的对象头部。这就使得在 Python 中，对对象的引用变得非常的统一，**我们只需要用一个 `PyObject*` 指针就可以引用任意的一个对象**。而不论该对象实际是一个什么对象。

图 1-1 显示了 Python 中不同对象与 `PyObject`、`PyVarObject` 在内存布局上的关系：

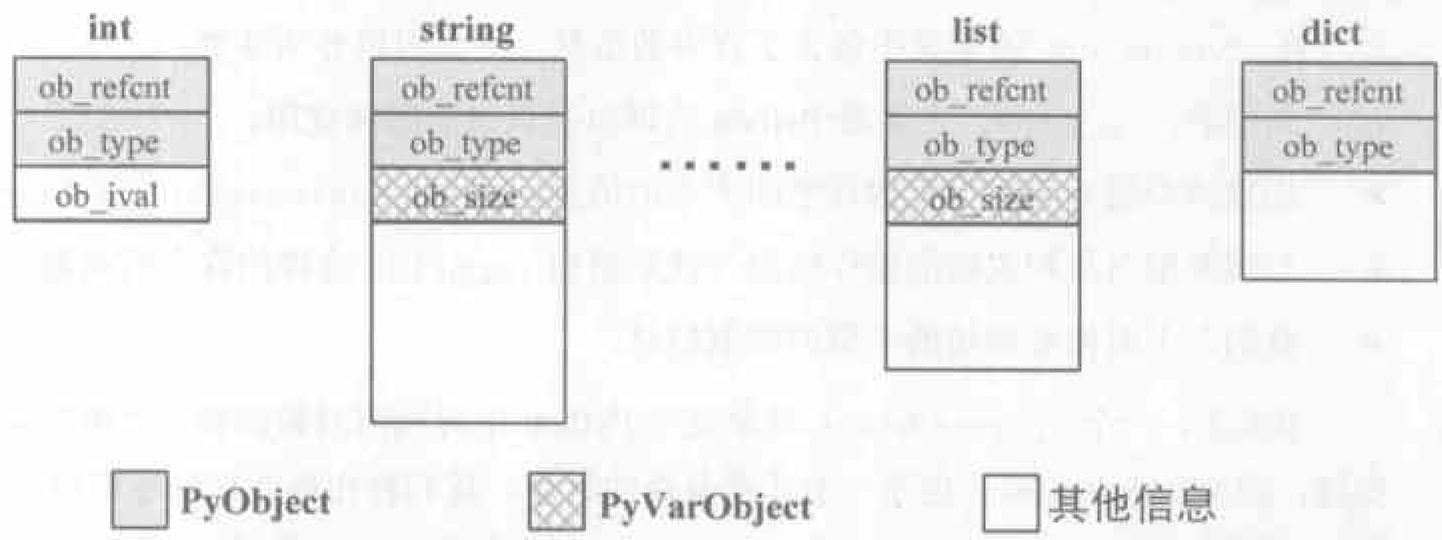



图 1-1 不同 Python 对象与 PyObject、PyVarObject 的关系

## 1.2 类型对象

在上面的描述中，我们看到了 Python 中所有对象共有信息的定义。所以，当内存中存在某一个 Python 对象时，该对象开始的几个字节的含义一定会符合我们的预想。但是，当我们顺着时间轴追溯，就会发现一个问题。当在内存中分配空间，创建对象的时候，毫无疑问地，必须要知道申请多大的空间。显然，这不会是一个定值，因为不同的对象，需要不同的空间，一个整数对象和一个字符串对象所需的空间肯定不同。那么，对象所需的内存空间的大小的信息到底在哪里呢？显然在 `PyObject` 中没有这样的信息。其实，这样

的说法是不对的，这个信息虽然不显见于 PyObject 的定义中，但它恰恰是隐身于 PyObject 之中。

实际上，占用内存空间的大小是对象的一种元信息，这样的元信息是与对象所属类型密切相关的，因此它一定会出现在与对象所对应的类型对象中。现在我们可以来详细考察一下类型对象_typeobject：

```
[object.h]
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */
    destructor tp_dealloc;
    printfunc tp_print;
    ……
    /* More standard operations (here for binary compatibility) */
    hashfunc tp_hash;
    ternaryfunc tp_call;
    ……
} PyTypeObject;
```

在_typeobject 的定义中包含了许多的信息，主要可以分为 4 类：

- 类型名，tp_name，主要是 Python 内部以及调试的时候使用；
- 创建该类型对象时分配内存空间大小的信息，即 tp_basicsize 和 tp_itemsize；
- 与该类型对象相关联的操作信息（就是诸如 tp_print 这样的许多的函数指针）；
- 我们在下面将要描述的类型的类型信息。

事实上，一个 PyTypeObject 对象就是 Python 中对面向对象理论中“类”这个概念的实现，而 PyTypeObject 也是一个非常复杂的话题，我们将在第 2 部分专门以一章的篇幅详细剖析构建在 PyTypeObject 之上的 Python 的类型和对象体系。这里仅仅是对 PyTypeObject 做一个粗略的介绍，如果读者有不太明白的地方，可以跳过，这并不影响第 1 部分的阅读。

## 1.2.1 对象的创建

考虑一下这个问题，假如我们命令 Python 创建一个整数对象，Python 内部究竟如何才能从无到有地创建出一个整数对象呢？一般来说，Python 会有两种方法。第一种是通过 Python C API 来创建，第二种是通过类型对象 PyInt_Type。

Python 对外提供了 C API，让用户可以从 C 环境中与 Python 交互，实际上，因为 Python 本身也是 C 写成的，所以 Python 内部也大量使用了这些 API。Python 的 C API 分成两类，

一类称为范型的 API，或者称为 AOL（Abstract Object Layer）。这类 API 都具有诸如 PyObject_***的形式，可以应用在任何 Python 对象身上，比如输出对象的 PyObject_Print，你可以 PyObject_Print(int object)，也可以 PyObject_Print(string object)，API 内部会有一整套机制确定最终调用的函数是哪一个。对于创建一个整数对象，我们可以采用如下的表达式：PyObject* intObj = PyObject_New(PyObject, &PyInt_Type)。

另一类是与类型相关的 API，或者称为 COL（Concrete Object Layer）。这类 API 通常只能作用在某一种类型的对象上，对于每一种内建对象，Python 都提供了这样的一组 API。比如对于整数对象，我们可以利用如下的 API 来创建，PyObject *intObj = PyInt_FromLong(10)，这样就创建了一个值为 10 的整数对象。

不论采用哪种 C API，Python 内部最终都是直接分配内存，因为 Python 对于内建对象是无所不知的。但是对于用户自定义的类型，比如通过 class A(object)，定义的一个类型 A，如果要创建 A 的实例对象（前面我们已经提到，实例对象可以视为面向对象理论中的"对象"概念），Python 就不可能事先提供 PyA_New 这样的 API。对于这种情况，Python 会通过 A 所对应的类型对象创建实例对象。图 1-2 给出了这样的一个例子：

```
>>> __builtins__.__dict__['int']
<type 'int'>
>>> a = int(10)
>>> a
10
>>> type(a)
<type 'int'>
>>>
>>> int.__base__
<type 'object'>
```

图 1-2 通过 PyInt_Type 创建整数对象

实际上，在 Python 完成运行环境的初始化之后，符号"int"就对应着一个表示为<type 'int'>的对象，这个对象其实就是 Python 内部的 PyInt_Type。当我们执行"int(10)"这样的表达式时，就是通过 PyInt_Type 创建了一个整数对象。

图 1-2 还显示出，在 Python 2.2 之后的 new style class 中，int 是一个继承自 object 的类型，类似于 int 对应着 Python 内部的 PyInt_Type，object 在 Python 内部则对应着 PyBaseObject_Type。图 1-3 显示了读者可能更熟悉的 C++中的定义 int 这种类型（class）的方式，以及在 Python 内部，这种继承的关系是如何实现的。

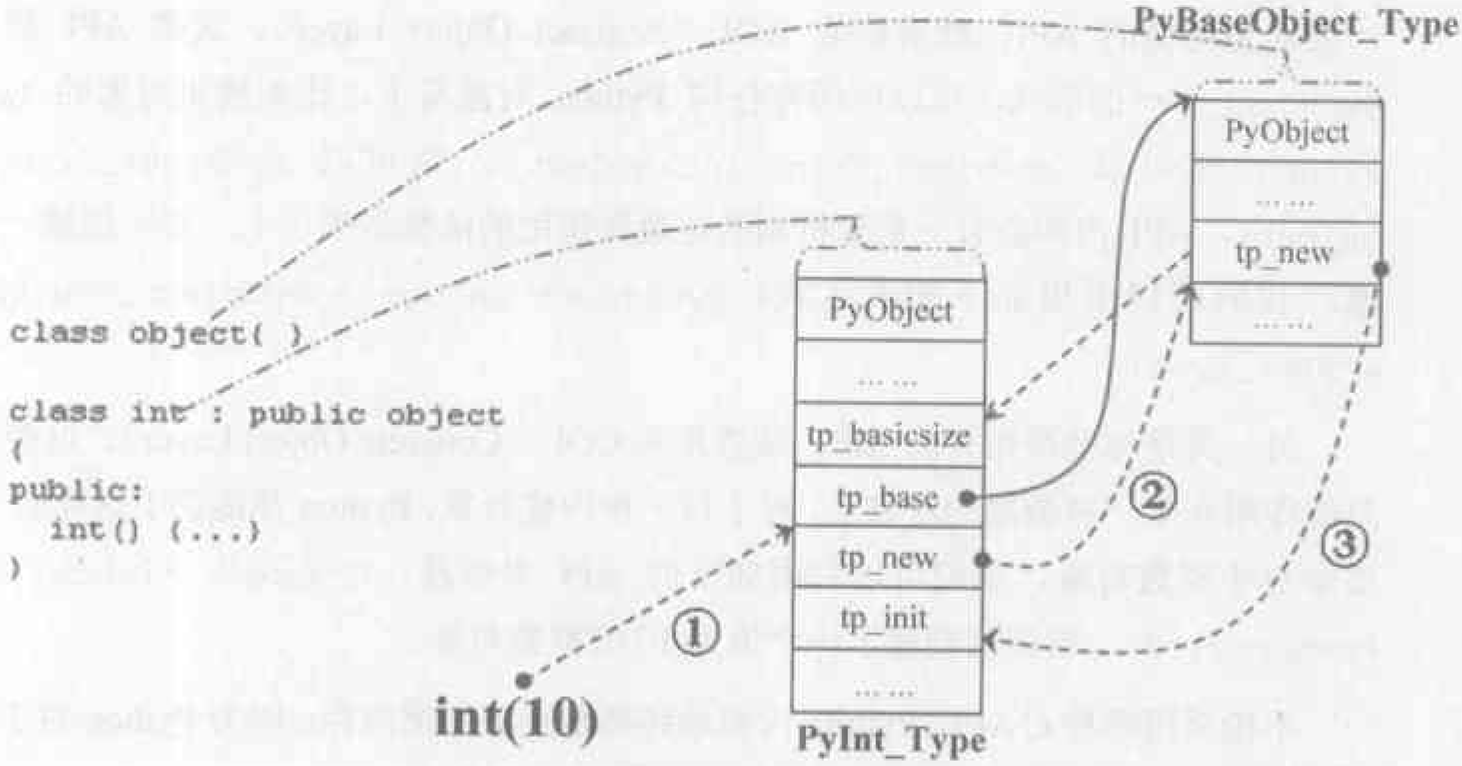



图 1-3 从 PyInt_Type 创建整数对象

标上序号的虚线箭头代表了创建整数对象的函数调用流程，首先 PyInt_Type 中的 tp_new 会被调用，如果这个 tp_new 为 NULL（真正的 PyInt_Type 中并不为 NULL，我们这里只是举例说明 tp_new 为 NULL 的情况），那么会到 tp_base 指定的基类中去寻找 tp_new 操作，PyBaseObject_Type 的 tp_new 指向了 object_new。在 Python 2.2 之后的 new style class 中，所有的类都是以 object 为基类的，所以最终会找到一个不为 NULL 的 tp_new。在 object_new 中，会访问 PyInt_Type 中记录的 tp_basicsize 信息，继而完成申请内存的操作。这个信息记录着一个整数对象应该占用多大内存，在 Python 源码中，你会看到这个值被设置成了 sizeof(PyIntObject)。在调用 tp_new 完成"创建对象"之后，流程会转向 PyInt_Type 的 tp_init，完成"初始化对象"的工作。对应到 C++中，tp_new 可以视为 new 操作符，而 tp_init 则可视为类的构造函数。

需要特别注意的是，这里只是以整数对象为例，说明类型对象在实例对象创建过程中的作用，实际上从下一章的分析中将会看到，作为 Python 内建对象的整数对象在创建时是有一些不同的。

## 1.2.2 对象的行为

在 PyTypeObject 中定义了大量的函数指针，这些函数指针最终都会指向某个函数，或者指向 NULL。这些函数指针可以视为类型对象中所定义的操作，而这些操作直接决定着一个对象在运行时所表现出的行为。

比如 PyTypeObject 中的 tp_hash 指明对于该类型的对象，如何生成其 hash 值。我

们看到 tp_hash 是一个 hashfunc 类型的变量，在 object.h 中，hashfunc 实际上是一个函数指针：typedef long (*hashfunc)(PyObject *)。在上一节中我们还看到了 tp_new，tp_init 是如何决定一个实例对象被创建出来并初始化的。在 PyTypeObject 中指定的不同的操作信息也正是一种对象区别于另一种对象的关键所在。

在这些操作信息中，有三组非常重要的操作族，在 PyTypeObject 中，它们是 tp_as_number、tp_as_sequence、tp_as_mapping。它们分别指向 PyNumberMethods、PySequenceMethods 和 PyMappingMethods 函数族，我们可以看一看 PyNumberMethods 这个函数族：

```
[object.h]
typedef PyObject * (*binaryfunc)(PyObject *, PyObject *);

typedef struct {
    binaryfunc nb_add;
    binaryfunc nb_subtract;
    ……
} PyNumberMethods;
```

在 PyNumberMethods 中，定义了作为一个数值对象应该支持的操作。如果一个对象能被视为数值对象，比如整数，整数对象当然是一个数值对象。那么在其对应的类型对象 PyInt_Type 中，tp_as_number.nb_add 就指定了对该对象进行加法操作时的具体行为。同样，PySequenceMethods 和 PyMappingMethods 中分别定义了作为一个序列对象和关联对象，应该支持的行为，这两种对象的典型例子是 list 和 dict。

对于一种类型来说，它完全可以同时定义三个函数族中的所有操作。换句话说，一个对象可以既表现出数值对象的特性，也可以表现出关联对象的特性。图 1-4 给出了这样一个例子：

```
>>> class MyInt(int):
        def __getitem__(self, key):
            return key + str(self)


>>> a = MyInt(1)
>>> b = MyInt(2)
>>> print a + b
3
>>> a['key']
'key1'
```

**图 1-4 数值对象和关联对象的混合体**

看上去 a['key'] 这样的操作是一个类似于 dict 这样的对象才会支持的操作。从 int 继承出来的 MyInt 应该自然就是一个数值对象，但是通过重写__getitem__这个 Python 中的 special method，可以视为指定了 MyInt 在 Python 内部对应的 PyTypeObject 对象的 tp_as_mapping.mp_subscript 操作。最终的结果是 MyInt 的实例对象可以“表现”得

像一个关联对象一样。归根结底就在于 PyTypeObject 中允许一种类型同时指定三种不同对象的行为特性。

## 1.2.3 类型的类型

仔细观察 PyTypeObject，会有一个有趣的发现。在 PyTypeObject 定义的最开始，可以发现 PyObject_VAR_HEAD，这意味着 Python 中的类型实际上也是一个对象。没错，在 Python 中，任何一个东西都是对象，而每一个对象都是对应一种类型的，那么一个有趣的问题就出现了，类型对象的类型是什么呢？这个问题听上去很绕口，实际上却非常重要，对于其他的对象，可以通过与其关联的类型对象确定其类型，那么通过什么来确定一个对象是类型对象呢？答案就是 PyType_Type：

```
[typeobject.c]
PyTypeObject PyType_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,                    /* ob_size */
    "type",                    /* tp_name */
    sizeof(PyHeapTypeObject),        /* tp_basicsize */
    sizeof(PyMemberDef),            /* tp_itemsize */
    ......
};
```

PyType_Type 在 Python 的类型机制中是一个至关重要的对象，所有用户自定义 class 所对应的 PyTypeObject 对象都是通过这个对象创建的。图 1-5 中显示了一般的 PyTypeObject 和 PyType_Type 的关系：

```
>>> class A(object):
        pass

>>> A.__class__
<type 'type'>
>>> int.__class__
<type 'type'>
>>> type.__class__
<type 'type'>
```

图 1-5 PyType_Type 与一般 PyTypeObject 的关系

图 1-5 中那个一再出现的<type 'type'>就是 Python 内部的 PyType_Type，它是所有 class 的 class，所以它在 Python 中被称为 metaclass。关于 PyType_Type 和 metaclass，这里不再深入阐述，在后面的章节中会详细剖析。

我们接着来看 PyInt_Type 是怎么和 PyType_Type 建立关系的。前面提到，在 Python 中，每一个对象都将自己的引用计数、类型信息保存在开始的部分中。为了方便对这部分内存的初始化，Python 中提供了几个有用的宏：

```
[object.h]
#ifdef Py_TRACE_REFS
```

```
    #define _PyObject_EXTRA_INIT 0, 0,
#else
    #define _PyObject_EXTRA_INIT
#endif

#define PyObject_HEAD_INIT(type)    \
   _PyObject_EXTRA_INIT        \
   1, type,
```

再回顾一下 PyObject 和 PyVarObject 的定义，初始化的动作就一目了然了。实际上，这些宏在各种内建类型对象的初始化中被大量地使用着。

以 PyInt_Type 为例，可以更清晰地看到一般的类型对象和这个特立独行的 PyType_Type 对象之间的关系：

```
[intobject.c]
PyTypeObject PyInt_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "int",
    sizeof(PyIntObject),
    ……
};
```

现在我们可以想象，看到一个整数对象在运行时的形象的表示了，如图 1-6 所示：

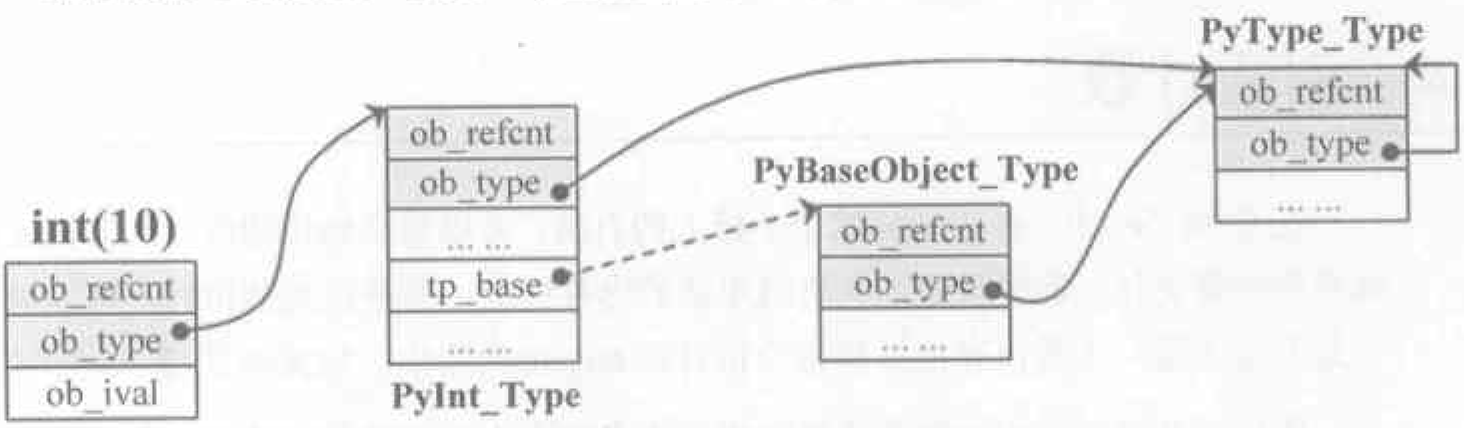



图 1-6 运行时整数对象及其类型之间的关系

# 1.3 Python 对象的多态性

通过 PyObject 和 PyTypeObject，Python 利用 C 语言完成了 C++所提供的对象的多态的特性。在 Python 创建一个对象，比如 PyIntObject 对象时，会分配内存，进行初始化。然后 Python 内部会用一个 PyObject*变量，而不是通过一个 PyIntObject*变量来保存和维护这个对象。其他对象也与此类似，所以在 Python 内部各个函数之间传递的都是一种范型指针——PyObject*。这个指针所指的对象究竟是什么类型的，我们不知道，只能从指针所指对象的 ob_type 域动态进行判断，而正是通过这个域，Python 实现了多态机制。

考虑下面的 Print 函数：

```
void Print(PyObject* object)
{
  object->ob_type->tp_print(object);
}
```

如果传给 Print 的指针是一个 PyIntObject*，那么它就会调用 PyIntObject 对象对应的类型对象中定义的输出操作，如果指针是一个 PyStringObject*，那么就会调用 PyStringObject 对象对应的类型对象中定义的输出操作。可以看到，这里同一个函数在不同情况下表现出了不同的行为，这正是多态的核心所在。

前面已经提到的 AOL 的 C API 正是建立在这中“多态”机制之上的。下面给出了一个简单的例子：

```
[object.c]
long PyObject_Hash(PyObject *v)
{
    PyTypeObject *tp = v->ob_type;
    if (tp->tp_hash != NULL)
        return (*tp->tp_hash)(v);
    ......
}
```

## 1.4 引用计数

在 C 或 C++中，程序员被赋予了极大的自由，可以任意地申请内存。但是权利的另一面则对应着责任，程序员必须负责将申请的内存释放，并释放无效指针。可以说，这一点正是万恶之源，大量内存泄露和悬空指针的 bug 由此而生，如黄河泛滥一发不可收拾。

现代的开发语言中一般都选择由语言本身负责内存的管理和维护，即采用了垃圾收集机制，比如 Java 和 C#。垃圾收集机制使开发人员从维护内存分配和清理的繁重工作中解放出来，但同时也剥夺了程序员与内存亲密接触的机会，并付出了一定的运行效率作为代价。现在看来，随着垃圾收集机制的完善，对时间要求不是非常高的程序完全可以通过使用垃圾收集机制的语言来完成，这部分程序占了现存的大多数的程序。这样做的好处是提高了开发效率，并降低了 bug 发生的几率。Python 同样也内建了垃圾收集机制，代替程序员进行繁重的内存管理工作，而引用计数正是 Python 垃圾收集机制的一部分。

Python 通过对一个对象的引用计数的管理来维护对象在内存中的存在与否。我们知道在 Python 中每一个东西都是一个对象，都有一个 ob_refcnt 变量。这个变量维护着该对象的引用计数，从而也最终决定着该对象的创建与消亡。

在 Python 中，主要是通过 Py_INCREF(op)和 Py_DECREF(op)两个宏来增加和减少一

个对象的引用计数。当一个对象的引用计数减少到 0 之后，Py_DECREF 将调用该对象的析构函数来释放该对象所占有的内存和系统资源。注意这里的“析构函数”借用了 C++的词汇，实际上这个析构动作是通过在对象对应的类型对象中定义的一个函数指针来指定的，就是那个 tp_dealloc。

如果熟悉设计模式中的 Observer 模式，就可以看到，这里隐隐约约透着 Observer 模式的影子。在 ob_refcnt 减为 0 之后，将触发对象销毁的事件。从 Python 的对象体系来看，各个对象提供了不同的事件处理函数，而事件的注册动作正是在各个对象对应的类型对象中静态完成的。

PyObject 中的 ob_refcnt 是一个 32 位的整形变量，这实际蕴含着 Python 所做的一个假设，即对一个对象的引用不会超过一个整形变量的最大值。一般情况下，如果不是恶意代码，这个假设显然是成立的。

需要注意的是，在 Python 的各种对象中，类型对象是超越引用计数规则的。类型对象“跳出三界外，不再五行中”，永远不会被析构。每一个对象中指向类型对象的指针不被视为对类型对象的引用。

在每一个对象创建的时候，Python 提供了一个_Py_NewReference(op)宏来将对象的引用计数初始化为 1。

在 Python 的源代码中可以看到，在不同的编译选项下(Py_REF_DEBUG，Py_TRACE_REFS)，引用计数的宏还要做许多额外的工作。下面展示的代码是 Python 在最终发行时这些宏所对应的实际的代码：

```
[object.h]
#define _Py_NewReference(op) ((op)->ob_refcnt = 1)
#define _Py_Dealloc(op) ((*(op)->ob_type->tp_dealloc)((PyObject *)(op)))
#define Py_INCREF(op) ((op)->ob_refcnt++)
#define Py_DECREF(op)                    \
    if (--(op)->ob_refcnt != 0)          \
        ;                \
    else                         \
        _Py_Dealloc((PyObject *)(op))

/* Macros to use in case the object pointer may be NULL: */
#define Py_XINCREF(op) if ((op) == NULL) ; else Py_INCREF(op)
#define Py_XDECREF(op) if ((op) == NULL) ; else Py_DECREF(op)
```

在一个对象的引用计数减为 0 时，与该对象对应的析构函数就会被调用，但是要特别注意的是，调用析构函数并不意味着最终一定会调用 free 释放内存空间，如果真是这样的话，那频繁地申请、释放内存空间会使 Python 的执行效率大打折扣（更何况 Python 已经多年背负了人们对其执行效率的不满）。一般来说，Python 中大量采用了内存对象池的技术，使用这种技术可以避免频繁地申请和释放内存空间。因此在析构时，通常都是将对

象占用的空间归还到内存池中。这一点在接下来对 Python 内建对象的实现中可以看得一清二楚。

## 1.5 Python 对象的分类

我们将 Python 的对象从概念上大致分为 5 类，需要指出的是，这种分类并不一定完全正确，不过是提供一种看待 Python 中对象的视角而已。

- Fundamental 对象：类型对象
- Numeric 对象：数值对象
- Sequence 对象：容纳其他对象的序列集合对象
- Mapping 对象：类似于 C++中 map 的关联对象
- Internal 对象：Python 虚拟机在运行时内部使用的对象

图 1-7 列出了我们的对象分类体系，并给出了每一个类别中的一些实例：

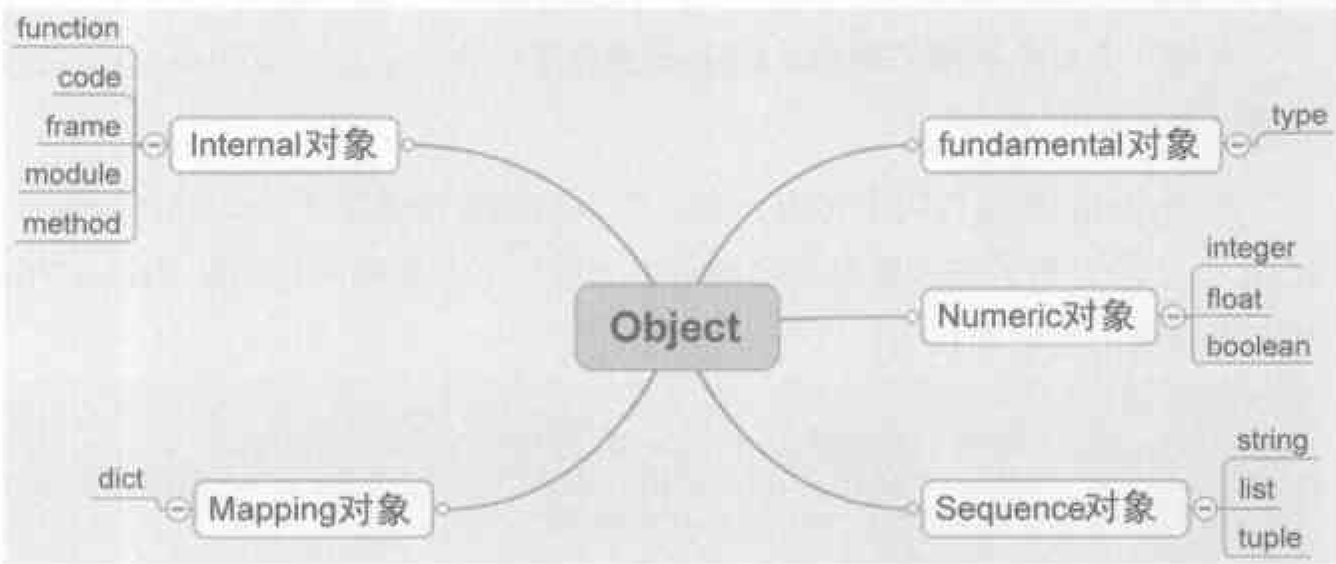



图 1-7 Python 中对象的分类

现在，我们已经有了一些关于 Python 对象体系的基本认识了，目前所掌握的这些认识已经足够我们支撑到细致剖析 Python 对象体系的那一天了。从现在开始，我们将正式进入本书的第一部分，剖析 Python 的内建对象。

CHAPTER

第 2 章

# Python 中的整数对象

在 Python 的所有对象中，整数对象是最简单的对象。从对 Python 对象机制的剖析来说，整数对象也最容易使读者真实地感受 Python 对象机制的切入点，因此我们对 Python 内建对象的剖析就从这个最简单的整数对象开始。

## 2.1 初识 PyIntObject 对象

Python 中对“整数”这个概念的实现是通过 `PyIntObject` 对象来完成的。在上一章初探 Python 对象体系时，我们看到了“定长对象”和“变长对象”的区别，这是一种对对象的二分法，实际上还存在着另一种对对象的二分法，这种分类法根据对象维护数据的可变性将对象分为“可变对象”（`mutable`）和“不可变对象”（`immutable`）。这章将要剖析的 `PyIntObject` 对象就是一个不可变对象，这种不变性是针对 `PyIntObject` 对象中所维护的那个真实的整数值而言的（在本节的描述中，我们将不刻意区分“`PyIntObject` 对象的值”和“`PyIntObject` 对象中维护的真实整数值”两种说法，而将其视为一致，都表示对象所维护的真实整数值）。也就是说，在创建了一个 `PyIntObject` 对象之后，就再也不能改变该对象的值了。在后续的章节中我们可以看到，这种不变性并非是 `PyIntObject` 对象所特有的性质，在 Python 中，除 `PyIntObject` 之外，还有很多对象也是不变对象，比如字符串对象等。

整数对象是如此简单，基于我们之前对 Python 对象机制的一般性剖析，闭上眼睛想象一下，似乎我们轻而易举就能搞定一个整数对象，对于它，似乎并不需要花费太多的笔墨。诚然，如果单纯地考虑一个静态的 `PyIntObject` 的实现，没有什么太困难的，然而当我们把目光投到运行时的整数对象身上，就会发现有许多值得深思的地方。

在 Python 的应用程序中，整数的使用是如此地广泛，其创生和湮灭又是如此频繁，考虑到 Python 所采用的引用计数机制，这是否意味着系统堆将面临着透过整数对象而涌来的狂风骤雨般的访问？这样的执行效率你可以接受吗？一旦引入运行时，我们就会看到，如何设计一个高效的机制，使得整数对象的使用不会成为 Python 的瓶颈，就成了一个必须面对的至关重要的设计决策，而解决方案也并非可以信手拈来。

整数对象池，这是 Python 给出的解答，我们将在本章中看到一个优雅而巧妙的整数对象的缓冲池机制。在此基础上，运行时的整数对象并非一个个独立的对象，而是如同自然界的蚂蚁一般，已经是通过一定的结构联结在一起的庞大的整数对象系统了。而这种面向特定对象的缓冲池机制也是 Python 语言实现时的核心设计策略之一，在后续的剖析中，我们会看到，几乎所有的内建对象，都会有自己所特有的对象池机制。

好，言归正传，在深入整数对象的运行时行为之前，我们再来回顾一下静态的整数对象的定义——PyIntObject：

```
[intobject.h]
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

Python 中的整数对象 PyIntObject 实际上就是对 C 中原生类型 long 的一个简单包装。从对 Python 对象机制的一般性描述中，我们知道，对于 Python 中的对象，与对象相关的元信息实际上都是保存在与对象对应的类型对象中的，对于 PyIntObject，这个类型对象是 PyInt_Type：

```
[intobject.c]
PyTypeObject PyInt_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "int",
    sizeof(PyIntObject),
    0,
    (destructor)int_dealloc,        /* tp_dealloc */
    (printfunc)int_print,           /* tp_print */
    0,                  /* tp_getattr */
    0,                  /* tp_setattr */
    (cmpfunc)int_compare,           /* tp_compare */
    (reprfunc)int_repr,         /* tp_repr */
    &int_as_number,             /* tp_as_number */
    0,                  /* tp_as_sequence */
    0,                  /* tp_as_mapping */
    (hashfunc)int_hash,         /* tp_hash */
    0,                  /* tp_call */
    (reprfunc)int_repr,         /* tp_str */
    PyObject_GenericGetAttr,        /* tp_getattro */
    0,                  /* tp_setattro */
    0,                  /* tp_as_buffer */
```

```
    Py_TPFLAGS_DEFAULT| Py_TPFLAGS_CHECKTYPES | Py_TPFLAGS_BASETYPE, /*
     tp_flags */
    int_doc,               /* tp_doc */
    0,                  /* tp_traverse */
    0,                  /* tp_clear */
    0,                  /* tp_richcompare */
    0,                  /* tp_weaklistoffset */
    0,                  /* tp_iter */
    0,                  /* tp_iternext */
    int_methods,          /* tp_methods */
    0,                  /* tp_members */
    0,                  /* tp_getset */
    0,                  /* tp_base */
    0,                  /* tp_dict */
    0,                  /* tp_descr_get */
    0,                  /* tp_descr_set */
    0,                  /* tp_dictoffset */
    0,                  /* tp_init */
    0,                  /* tp_alloc */
    int_new,               /* tp_new */
    (freefunc)int_free,               /* tp_free */
};
```

这里完整地列出了 PyIntObject 对象的元信息，这是我们第一次，也是最后一次花费如此的篇幅来展示一个对象的元信息，在以后的章节中，我们将只会给出与某个主题相关的元信息。

在 PyInt_Type 中保存了关于 PyIntObject 对象的丰富元信息，其中有 PyIntObject 对象应该占用的内存大小，PyIntObject 对象的文档信息，而更多的是 PyIntObject 对象所支持的操作。在表 2-1 中，列出了一些 PyIntObject 所支持的操作：

表 2-1

| | |
|---|---|
| int_dealloc | PyIntObject 对象的析构操作 |
| int_free | PyIntObject 对象的释放操作 |
| int_repr | 转化成 PyStringObject 对象 |
| int_hash | 获得 HASH 值 |
| int_print | 打印 PyIntObject 对象 |
| int_compare | 比较操作 |
| int_as_number | 数值操作集合 |
| int_methods | 成员函数集合 |

下面是一个例子，我们可以看一看如何比较两个整数对象的大小。

```
[intobject.c]
static int int_compare(PyIntObject *v, PyIntObject *w)
{
    register long i = v->ob_ival;
    register long j = w->ob_ival;
```

```
    return (i < j) ? -1 : (i > j) ? 1 : 0;
}
```

显然，PyIntObject 对象的比较操作实际上就是简单地将它所维护的 long 值进行比较。在 PyInt_Type 这个元信息集合中，需要特别注意的是 int_as_number 这个域：

```
[intobject.c]
static PyNumberMethods int_as_number = {
    (binaryfunc)int_add,    /*nb_add*/
    (binaryfunc)int_sub,    /*nb_subtract*/
    ……
    (binaryfunc)int_div,    /* nb_floor_divide */
    int_true_divide,    /* nb_true_divide */
    0,          /* nb_inplace_floor_divide */
    0,          /* nb_inplace_true_divide */
};
```

上一章已经提到，这个 PyNumberMethods 中定义了一个对象作为数值对象时的所有可选的操作信息。在 Python 2.5 中，PyNumberMethods 中一共有 39 个函数指针，即其中定义了 39 种可选的操作，这 39 种操作包括加法、减法、乘法、模运算等。

在 int_as_number 中，确定了对于一个整数对象，这些数值操作应该如何进行。当然，在 PyNumberMethods 的 39 种数值操作中，并非所有的操作都要求一定要被实现。比如在 int_as_number 中，就可以看到，有相当多的操作是没有实现的。作为一个数值操作的例子，我们可以看一下 PyIntObject 中加法操作是如何实现的（见代码清单 2-1）。

**代码清单 2-1**

```
[intobject.h]
//宏，牺牲类型安全，换取执行效率
#define PyInt_AS_LONG(op) (((PyIntObject *)(op))->ob_ival)

[intobject.c]
#define CONVERT_TO_LONG(obj, lng)       \
    if (PyInt_Check(obj)) {         \
        lng = PyInt_AS_LONG(obj);    \
    }               \
    else {          \
        Py_INCREF(Py_NotImplemented);    \
        return Py_NotImplemented;    \
    }

static PyObject* int_add(PyIntObject *v, PyIntObject *w)
{
    register long a, b, x;
    CONVERT_TO_LONG(v, a);
    CONVERT_TO_LONG(w, b);
    x = a + b;
    //[1] : 检查加法结果是否溢出
    if ((x^a) >= 0 || (x^b) >= 0)
        return PyInt_FromLong(x);
    return PyLong_Type.tp_as_number->nb_add((PyObject *)v, (PyObject *)w);
}
```

如你所想，PyIntObject 对象所实现的加法操作是直接在其维护的 long 值上进行的，可以看到，在完成了加法操作后，代码清单 2-1 的[1]处还进行了溢出的检查。如果没有溢出，就返回一个新的 PyIntObject，这个 PyIntObject 所拥有的值正好是加法操作的结果。

在 Python 的实现中，对某些会频繁执行的代码，都会同时提供函数和宏两种版本，比如在上面列出的代码中的 PyInt_AS_LONG，与之对应的还有一个函数 PyInt_AsLong。宏版本的 PyInt_AS_LONG 可以省去一次函数调用的开销，但是其牺牲了类型安全，因为其参数 op 完全可以不是一个 PyIntObject 对象，比如程序员在使用 C 编写 Python 的扩展模块时，可能由于疏忽导致这样的错误。而查看 intobject.c 中的函数版 PyInt_AsLong，则会多方检查类型安全性，当然，这就以执行效率作为代价了。

从 PyIntObject 对象的加法操作的实现可以清晰地看到，PyIntObject 确实是一个 immutable 的对象，因为在操作完成之后，原来参与操作的任何一个对象都没有发生改变，取而代之的是一个全新的 PyIntObject 对象于虚无中诞生。

如果加法的结果有溢出，那么结果就再不是一个 PyIntObject 对象，而是一个 PyLongObject 对象了。图 2-1 对 PyIntObject 对象 a 执行 a+a 的操作时，就会引起加法结果溢出，从而返回一个 PyLongObject 对象：

```
>>> a = 0x7fffffff
>>> type(a)
<type 'int'>
>>> type(a+a)
<type 'long'>
```

图 2-1 加法溢出的例子

另一个有趣的元信息是 PyIntObject 对象的文档信息，这个元信息维护在 int_doc 域中。文档无缝地集成在语言的实现中，这一点，是 Python 相对于其他语言的一大特点。我们可以在 Python 的交互环境下通过 PyIntObject 对象的__doc__属性看到 int_doc 维护的文档，如图 2-2 所示：

```
>>> a = 1
>>> print a.__doc__
int(x[, base]) -> integer

Convert a string or number to an integer, if possible
argument will be truncated towards zero (this does no
representation of a floating point number!)  When con
the optional base.  It is an error to supply a base w
non-string. If the argument is outside the integer ra
will be returned instead.
```

图 2-2 整数的文档信息

```
[python.h]
#define PyDoc_VAR(name) static char name[]
#define PyDoc_STRVAR(name,str) PyDoc_VAR(name) = PyDoc_STR(str)
#ifdef WITH_DOC_STRINGS
```

```
#define PyDoc_STR(str) str
#else
#define PyDoc_STR(str) ""
#endif
```

```
[intobject.c]
PyDoc_STRVAR(int_doc,
"int(x[, base]) -> integer\n\
\n\
Convert a string or number to an integer, if possible.  A floating point\n\
argument will be truncated towards zero (this does not include a string\n\
representation of a floating point number!)  When converting a string, use\n\
the optional base.  It is an error to supply a base when converting a\n\
non-string. If the argument is outside the integer range a long object\n\
will be returned instead.");
```

## 2.2 PyIntObject 对象的创建和维护

### 2.2.1 对象创建的 3 种途径

在上一章我们已经提到，Python 中创建一个实例对象可以通过 Python 暴露的 C API，也可以通过类型对象完成创建动作。在 Python 自身的实现中，几乎都是调用 C API 来创建内建实例对象的。而且，对于内建对象，即便是通过内建类型对象中的 tp_new、tp_init 操作创建实例对象，实际上最终还是会调用 Python 为特定内建对象准备的 C API。所以在本书第一部分对内建实例对象的创建之分析中，我们将把分析的重点都放在 Python 为这种内建实例对象暴露出来的 C API 上。无论通过哪种方式创建内建实例对象，我们分析所得出的结论都是正确的。

在 intobject.h 中可以看到，为了创建一个 PyIntObject 对象，Python 提供了 3 条途径：

```
   PyObject *PyInt_FromLong(long ival)
   PyObject*  PyInt_FromString(char *s, char **pend, int base)
#ifdef Py_USING_UNICODE
   PyObject*PyInt_FromUnicode(Py_UNICODE *s, int length, int base)
#endif
```

分别是从 long 值，从字符串以及 Py_UNICODE 对象生成 PyIntObject 对象。在这里我们只考察从 long 值生成 PyIntObject 对象。因为 PyInt_FromString 和 PyInt_FromUnicode 实际上都是先将字符串或 Py_UNICODE 对象转换成浮点数，然后再调用 PyInt_FromFloat。如此看来，PyInt_FromString 和 PyInt_FromUnicode 不过利用了设计模式中 Adaptor Pattern 的思想对整数对象的核心创建函数 PyInt_FromFloat 进行了接口转换罢了：

```
[intobject.c]
PyObject*  PyInt_FromString(char *s, char **pend, int base)
{
    char *end;
    long x;
    ......
    //将字符串转换为 long 值
    if (base == 0 && s[0] == '0')
    {
        x = (long) PyOS_strtoul(s, &end, base);
    }
    else
        x = PyOS_strtol(s, &end, base);
    ......
    return PyInt_FromLong(x);
}
```

为了深刻地理解 PyIntObject 对象的创建过程，首先必须要深入了解 Python 中整数对象在内存中的组织方式。前面我们已经提到，在运行期间，一个个的整数对象在内存中并不是独立存在，单兵作战的，而是形成了一个整数对象系统。我们首先就重点考察一下 Python 中整数对象系统的结构。

## 2.2.2 小整数对象

在实际的编程中，数值比较小的整数，比如 1、2、29 等，可能在程序中会非常频繁地使用。想一想 C 语言中的 for 循环，就可以了解为什么这些小整数会有那么频繁的使用场合。在 Python 中，所有的对象都存活在系统堆上。这就是说，如果没有特殊的机制，对于这些频繁使用的小整数对象，Python 将一次又一次地使用 malloc 在堆上申请空间，并且不厌其烦地一次次 free。这样的操作不仅大大降低了运行效率，而且会在系统堆上造成大量的内存碎片，严重影响 Python 的整体性能。

显然，Guido 是决不能容许这样的方案存在的，于是在 Python 中，对小整数对象使用了对象池技术。刚才我们说了，PyIntObject 对象是不可变对象，这个信息预示着一个天大的喜讯，即对象池里的每一个 PyIntObject 对象都能够被任意地共享。

给你一个整数 100，它是一个“小”整数吗？那么 101 呢？小整数和大整数的分界点在哪里？Python 的回答是：你的地盘你做主，你想这个分界点在哪里它就在哪里。

现在，我们有一个好消息和一个坏消息。好消息是，Python 确实提供了一种方法，通过这种方法，用户可以调整小整数与大整数的分界点，从而动态确定小整数对象池中到底应该有多少个小整数对象。呃，但是，老实说，坏消息可能会打击你的积极性，Python 提供的这种方法非常原始，为了达到动态调整的目的，你只有自己修改源代码，然后重新编译出新的 Python 来。

```
[intobject.c]
#ifndef NSMALLPOSINTS
    #define NSMALLPOSINTS        257
#endif
#ifndef NSMALLNEGINTS
    #define NSMALLNEGINTS        5
#endif
#if NSMALLNEGINTS + NSMALLPOSINTS > 0
    static PyIntObject *small_ints[NSMALLNEGINTS + NSMALLPOSINTS];
#endif
```

“大隐隐于市，小隐隐于野”，在 intobject.c 超过 1000 行的代码中，这个毫不起眼的 small_ints 就是举足轻重的小整数对象的对象池，准确地说，应该是一个 PyIntObject* 池，不过没有关系，我们就打算以小整数对象池来称呼它。在 Python 2.5 中，将小整数集合的范围默认设定为[-5，257)。但是你完全可以修改 NSMALLPOSINTS 和 NSMALLNEGINTS，重新编译 Python，从而将这个范围向两端伸展或收缩。

对于小整数对象，Python 直接将这些整数对应的 PyIntObject 缓存在内存中，并将其指针存放在 small_ints 中。那么对于大整数呢？很显然，整数对象是编程中使用得非常多的东西，谁敢拍胸脯保证只有小整数才会被频繁地使用呢。如果将所有的整数对应的 PyIntObject 对象都缓存在内存池中，自然是再理想不过了，但是这样对内存的使用是会被视为败家子的，难免遭人鄙视。开放源码是放在公共空间供无数人检阅批判的，恐怕即便胆大如牛者，也不敢在内存使用上出此下策吧。时间与空间的两难选择，这个计算机领域最基本的矛盾就在这里浮出水面了。

## 2.2.3 大整数对象

Python 的设计者们所做出的妥协是，对于小整数，在小整数对象池中完全地缓存其 PyIntObject 对象。而对其他整数，Python 运行环境将提供一块内存空间，这些内存空间由这些大整数轮流使用，也就是说，谁需要的时候谁就使用。这样免去了不断地 malloc 之苦，又在一定程度上考虑了效率问题。下面将详细剖析其实现机制。

在 Python 中，有一个 PyIntBlock 结构，在这个结构的基础上，实现了的一个单向列表。

```
[intobject.c]
#define BLOCK_SIZE  1000    /* 1K less typical malloc overhead */
#define BHEAD_SIZE  8   /* Enough for a 64-bit pointer */
#define N_INTOBJECTS    ((BLOCK_SIZE - BHEAD_SIZE) / sizeof(PyIntObject))

struct _intblock {
    struct _intblock *next;
    PyIntObject objects[N_INTOBJECTS];
};
```

```
typedef struct _intblock PyIntBlock;

static PyIntBlock *block_list = NULL;
static PyIntObject *free_list = NULL;
```

PyIntBlock，顾名思义，就是说这个结构里维护了一块内存（block），其中保存了一些 PyIntObject 对象。从 PyIntBlock 的定义中可以看到，在一个 PyIntBlock 中维护着 N_INTOBJECTS 个对象，做一个简单的计算，就可以知道是 82 个。显然，这个地方也是 Python 的设计者留给你的可以动态调整的地方，不过，你需要再一次地修改源代码并重新编译。

PyIntBlock 的单向列表通过 block_list 维护，每一个 block 中都维护了一个 PyIntObject 数组——objects，这就是真正用于存储被缓存的 PyIntObject 对象的内存。可以想象，在 Python 运行的某个时刻，某个 block 的 objects 中，有一些内存已经被使用了，而另一些内存则处于空闲状态。这些空闲状态的内存必须组织起来，这样，Python 在需要新的内存来存储新的 PyIntObject 对象时，才能快速获得所需的内存。Python 使用一个单向链表来管理全部 block 的 objects 中所有的空闲内存，这个自由内存链表的表头就是 free_list。当然，最开始的时候，这两个指针都被设置为空指针，如图 2-3 所示。




图 2-3 free_list 和 block_list 的初始状态

注：在此后的图示中，我们将统一用实线菱尾箭头表示 block_list，虚线菱尾箭头表示 free_list。

## 2.2.4 添加和删除

好了，现在大体上可以想象 Python 中整数对象系统在内存中是一种怎样的结构了。下面我们将通过对 PyInt_FromLong 进行细致入微的考察，真实地展现一个个 PyIntObject 对象是如何从无到有地产生，并融入到 Python 的整数对象系统中的（见代码清单 2-2）。

**代码清单 2-2**

```
[intobject.c]
PyObject* PyInt_FromLong(long ival)
{
    register PyIntObject *v;
#if NSMALLNEGINTS + NSMALLPOSINTS > 0
    //[1] : 尝试使用小整数对象池
    if (-NSMALLNEGINTS <= ival && ival < NSMALLPOSINTS) {
        v = small_ints[ival + NSMALLNEGINTS];
        Py_INCREF(v);
        return (PyObject *) v;
    }
```

```
#endif
    //[2] : 为通用整数对象池申请新的内存空间
    if (free_list == NULL) {
        if ((free_list = fill_free_list()) == NULL)
            return NULL;
    }
    //[3] : (inline)内联 PyObject_New 的行为
    v = free_list;
    free_list = (PyIntObject *)v->ob_type;
    PyObject_INIT(v, &PyInt_Type);
    v->ob_ival = ival;
    return (PyObject *) v;
}
```

PyIntObject 对象的创建通过两步完成：

- 如果小整数对象池机制被激活，则尝试使用小整数对象池；
- 如果不能使用小整数对象池，则使用通用的整数对象池。

#### 2.2.4.1 使用小整数对象池

如果 NSMALLNEGINTS + NSMALLPOSINTS > 0 成立，那么 Python 认为小整数对象池机制被激活了，PyInt_FromLong 会首先在代码清单 2-2 的[1]处检查传入的 long 值是否属于小整数的范围，如果确实属于小整数，一切就变得简单了，只需要返回在小整数对象池中的对应的对象就可以了。

如果小整数对象池机制没有被激活，或者传入的 long 值不是属于小整数，Python 就会转向由 block_list 维护的通用整数对象池。正如前面我们所描述的，Python 需要在某块 block 的 objects 中，寻找一块可用于存储新的 PyIntObject 对象的内存，于是，重任就落在了 free_list 身上。

#### 2.2.4.2 创建通用整数对象池

显然，当首次调用 PyInt_FromLong 时，free_list 必定为 NULL，这时 Python 会在代码清单 2-2 的[2]处调用 fill_free_list，创建新的 block，从而也就创建了新的空闲内存。需要注意的是，Python 对 fill_free_list 的调用不光会发生在对 PyInt_FromLong 的首次调用时，在 Python 运行期间，只要所有 block 的空闲内存被都使用完了，就会导致 free_list 变为 NULL，从而在下一次 PyInt_FromLong 的调用时激发对 fill_free_list 的调用。

**代码清单 2-3**

```
[intobject.c]
static PyIntObject* fill_free_list(void)
{
```

```
    PyIntObject *p, *q;
    //[1] : 申请大小为sizeof(PyIntBlock)的内存空间，并链接到已有的block list中
    p = (PyIntObject *) PyMem_MALLOC(sizeof(PyIntBlock));
    ((PyIntBlock *)p)->next = block_list;
    block_list = (PyIntBlock *)p;
    //[2] : 将PyIntBlock中的PyIntObject数组——objects——转变成单向链表
    p = &((PyIntBlock *)p)->objects[0];
    q = p + N_INTOBJECTS;
    while (--q > p)
        q->ob_type = (struct _typeobject *)(q-1);
    q->ob_type = NULL;
    return p + N_INTOBJECTS - 1;
}
```

在 `fill_free_list` 中，会首先在代码清单 2-3 的[1]处申请一个新的 PyIntBlock 结构，如图 2-4 所示：

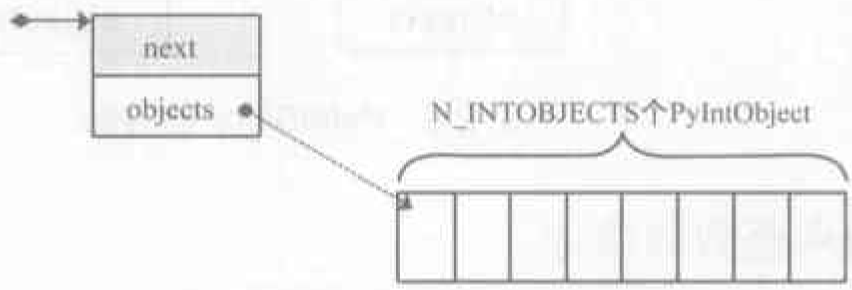



图 2-4 PyIntBlock 的内存布局

注：图中的虚线并不表示指针关系，虚线表示 objects 的更详细的表示方式，下同。

这时 block 中的 `objects` 还仅仅是一个 `PyIntObject` 对象的数组，接下来，Python 将 objects 中的所有 `PyIntObject` 对象通过指针依次连接起来，从而将数组转变成一个单向链表，这就是代码清单 2-3 的[2]处的行为。Python 从 `objects` 数组的最后一个元素开始链接的过程，在整个链接过程中，Python 使用了 `PyObject` 中的 `ob_type` 指针作为连接指针。可以看到，Python 的设计者为了解决问题，在某个局部点，放弃了对类型安全的坚持，如同政治一样，程序的开发也是一门妥协的艺术☺。

图 2-5 展示了代码清单 2-3 的[2]处的链表转换动作完成之后的 block，其中用虚线箭头展示了[2]开始时 `p` 和 `q` 的初始状态。可以看到，当链表转换完成之后，`free_list` 也在它该出现的位置了。从 `free_list` 开始，沿着 `ob_type` 指针，就可以遍历刚刚创建的 PyIntBlock 中的所有空闲的为 `PyIntObject` 准备的内存了。

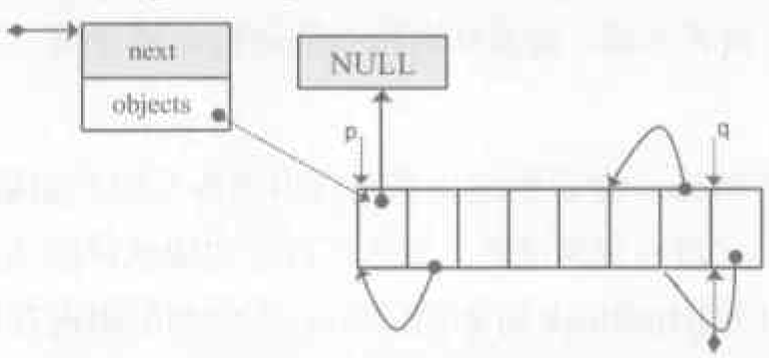



图 2-5 完成链表转换过程后的 PyIntBlock

当一个 block 中还有剩余的内存没有被一个 PyIntObject 占用时，free_list 就不会指向 NULL。所以在这种情况下调用 PyInt_FromLong 不会申请新的 block。只有在所有 block 中的内存都被占用了，PyInt_FromLong 才会再次调用 fill_free_list 申请新的空间，为新的 PyIntObject 创造新的家园。

前面我们提到，Python 是通过 block_list 来维护整个整数对象的通用对象池的。显然，这里新创建的 block 也必须加入到 block_list 所维护的链表中，这个链入的动作在代码清单 2-3 的[1]处完成。作为一个例子，图 2-6 显示了两次申请 block 后 block_list 所维护的链表的情况。值得注意的是，block_list 始终是指向最新创建的 PyIntBlock 对象。

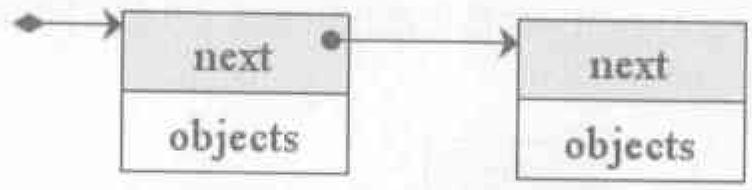



图 2-6 PyIntBlock 的链表

### 2.2.4.3 使用通用整数对象池

在 PyInt_FromLong 中，在必要的空间被申请之后，Python 会从当前由 free_list 所维护的自由内存链表中划出一块，并在这块内存上创建所需要的新的 PyIntObject 对象，同时，还会对 PyIntObject 对象完成必要的初始化工作。当然，Python 还将调整 free_list 指针，使其指向下一块还没有被使用的内存。

在图 2-6 中我们发现，两个 PyIntBlock 处于同一个链表当中，但是每一个 PyIntBlock 中至关重要的存放 PyIntObject 对象的 objects 却是分离的。这样的结构存在着隐患，考虑这样的情况：

现在有两个 PyIntBlock 对象，PyIntBlock1 和 PyIntBlock2，PyIntBlock1 中的 objects 已经被 PyIntObject 对象填满，而 PyIntBlock2 种的 object 只填充了一部分。所以现在 free_list 指针指向的是 PyIntBlock2.objects 中空闲的内存块。假设现在 PyIntBlock1.objects 中的一个 PyIntObject 对象被删除了，这意味着 PyIntBlock1 中出现了一块空闲的内存，那么下次创建新的 PyIntObject 对象时应该使用 PyIntBlock1 中的这块内存。倘若不然，就意味着所有的内存只能使用一次，这跟内存泄漏也就没什么区别了。

如何才能使 Python 意识到这块重获自由的内存呢？如果像图 2-6 所示的 PyIntBlock 对象间的 objects 没有任何联系，显然不可能实现这样的功能，所以它们之间一定存在联系。实际上，不同 PyIntBlock 对象的 objects 中的空闲内存块是被链接在一起的，形成了一个单向链表，指向表头的指针正是 free_list。

那么，不同 PyIntBlock 中的空闲内存块是在什么时候被链接在一起的呢，这一切都发生在一个 PyIntObject 对象被销毁的时候。

列位看官，花开两朵，各表一枝☺，这里我们先放下自由内存链表，仔细考察一下一个 PyIntObject 对象在被销毁时都发生了什么事。在对 Python 中对象机制的分析中，我们已经看到，每一个对象都有一个引用计数与之相关联，当这个引用计数减少到 0 时，就意味着这个世上再也没有谁需要它了，于是 Python 会负责将这个对象销毁。Python 中不同对象在销毁时会进行不同的动作，销毁动作在与对象对应的类型对象中被定义，这个关键的操作就是类型对象中的 tp_dealloc。

下面看一看 PyIntObject 对象的 tp_dealloc 操作：

```
[intobject.c]
static void int_dealloc(PyIntObject *v)
{
    if (PyInt_CheckExact(v)) {
        v->ob_type = (struct _typeobject *)free_list;
        free_list = v;
    }
    else
        v->ob_type->tp_free((PyObject *)v);
}
```

在前面我们说了，由 block_list 维护的 PyIntBlock 链表中的内存实际是所有的大整数对象共同分享的。俗话说，皇帝轮流坐，明年到我家。当一个 PyIntObject 对象被销毁时，它所占有的内存并不会被释放，归还给系统，而是继续被 Python 保留着。但是这一块内存在整数对象被销毁后变为了自由内存，将来可供别的 PyIntObject 使用，所以 Python 应该将其链入了 free_list 所维护的自由内存链表。int_dealloc 完成的就是这么一个简单的指针维护的工作。当然，这些动作是在销毁的对象确实是一个 PyIntObject 对象时发生的。如果删除的对象是一个整数的派生类的对象，那么 int_dealloc 不做任何动作，只是简单地调用派生类型中指定的 tp_free。

在图 2-7 中我们相继创建和删除 PyIntObject 对象，并展示了这一过程中，内存中的 PyIntObject 对象以及 free_list 指针的变化情况。请注意，在 Python 的实际行为中，创建 2、3、4 这样的整数对象，使用的实际上是 small_ints 这个小整数对象池，在这里我们的目的仅仅是为了展示通用整数对象池的动态变化，没有考虑 2、3、4 实际使用的内存。

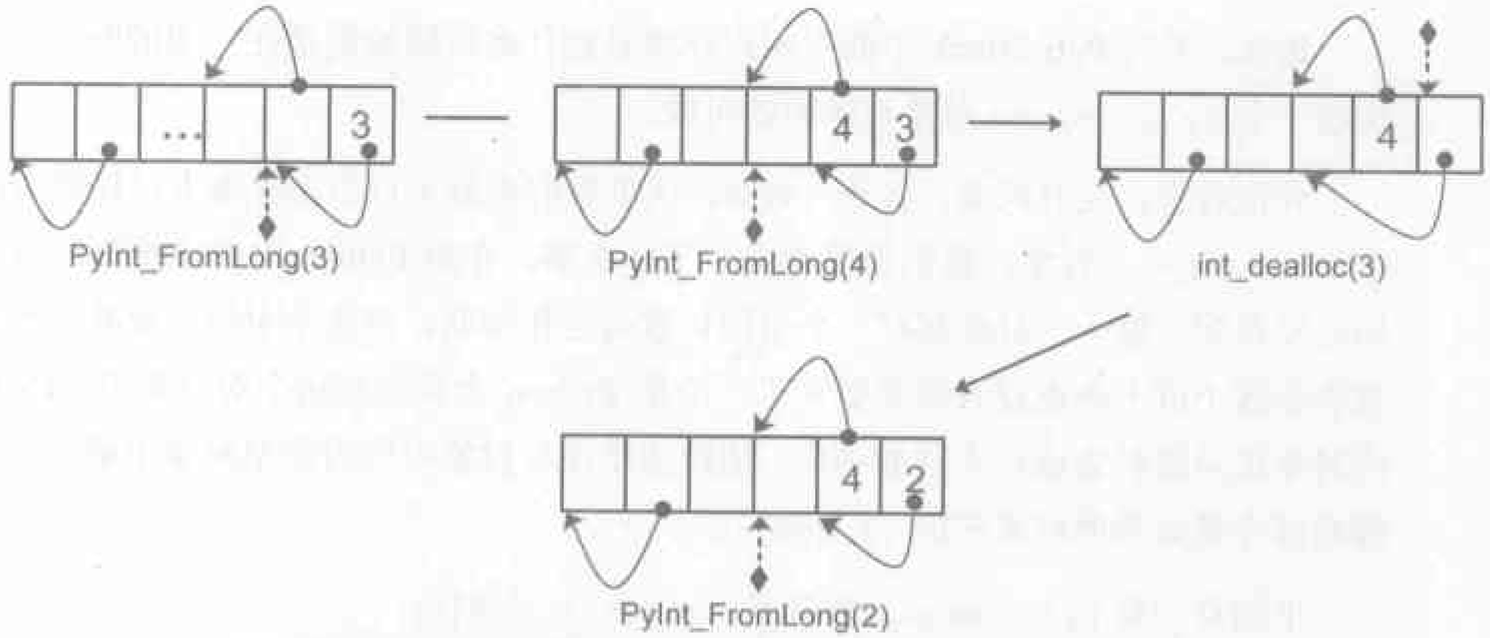



图 2-7 创建和删除 PyIntObject 对象时缓冲池的变化

现在回头来看看刚才提到的不同 PyIntBlock 对象的 objects 间的空闲内存的互连问题。其实很简单，不同 PyIntBlock 对象中空闲内存的互连也是在 int_dealloc 被调用时实现的。图 2-8 展示了这个过程（白色表示空闲内存）。

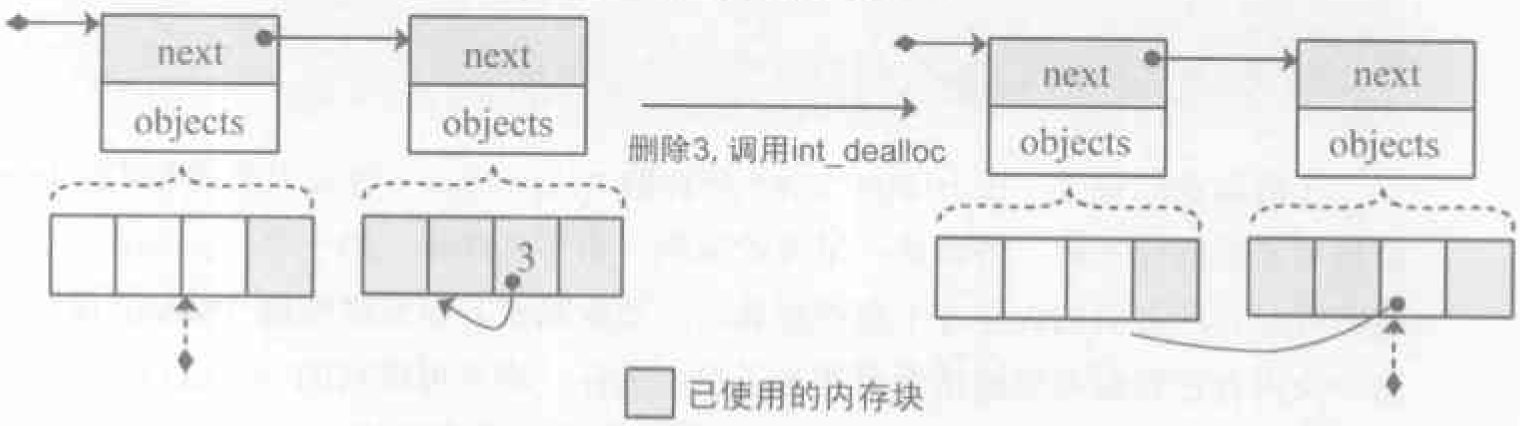



图 2-8 不同 PyIntBlock 中的空闲内存的互连

图 2-8 中隐藏着 Python 的整数对象系统实现中的一个惊天隐秘，细心的读者一定发现了，当一个整数对象的引用计数变为 0 时，就会被 Python 回收，但是在 int_dealloc 中，正如图 2-8 所示，仅仅是将该整数对象的内存重新加入到自由内存链表中。也就是说，在 int_dealloc 中，永远不会向系统堆交还任何内存。一旦系统堆中的某块内存被 Python 申请用于整数对象，那么这块内存在 Python 结束之前，永远不会被释放。这听上去跟内存泄漏极为相似，尽管从 Python 的通用整数对象池的实现我们可以看到，由于内存共享，Python 用于实现该对象池的内存与历史上创建的整数对象的个数无关，而仅仅与同一时刻共存的整数对象个数的最大值有关，但是，从理论上，我们仍然可以利用这一漏洞将系统的内存全部吃光。做一个简单的计算就可以看到这一点，一个 PyIntObject 对象的大小是 12 个字节，假如系统有 1GB 内存，那么这 1GB 的内存仅仅可容纳从 0 到 89 478 486 的整数对象，坦白说，这个值，对于希望利用这一实现漏洞的人来说，并不大。

## 2.2.5 小整数对象池的初始化

现在，关于 Python 的整数对象系统，还剩下最后一个问题了。在 small_ints 中（如图 2-9 所示），我们看到，它维护的只是 PyIntObject 的指针，那么这些与天地同寿的小整数对象是在什么地方被创建和初始化的呢。完成这一切的神秘的函数正是_PyInt_Init。

```
[intobject.c]
int _PyInt_Init(void)
{
    PyIntObject *v;
    int ival;
#if NSMALLNEGINTS + NSMALLPOSINTS > 0
    for (ival = -NSMALLNEGINTS; ival < NSMALLPOSINTS; ival++)
    {
        if (!free_list && (free_list = fill_free_list()) == NULL)
            return 0;
        //内联(inline)PyObject_New的行为
        v = free_list;
        free_list = (PyIntObject *)v->ob_type;
        PyObject_INIT(v, &PyInt_Type);
        v->ob_ival = ival;
        small_ints[ival + NSMALLNEGINTS] = v;
    }
#endif
    return 1;
}
```

从小整数的创建过程中可以看到，这些永生不灭的小整数对象也是生存在由 block_list 所维护的内存上的。在 Python 初始化的时候，_PyInt_Init 被调用，内存被申请，小整数对象被创建，然后就仙福永享，寿与天齐了☺。

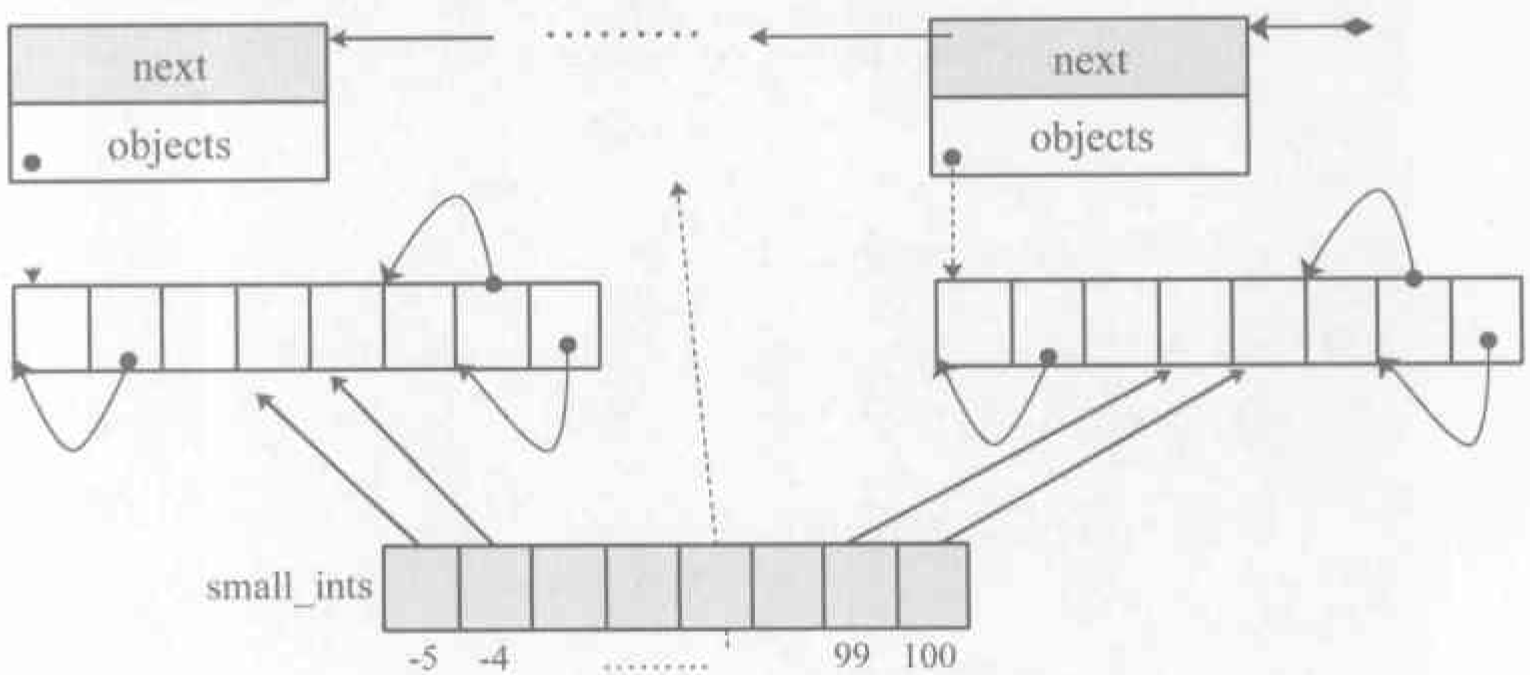



图 2-9 初始化完成之后的 small_ints

# 2.3 Hack PyIntObject

现在，让我们荡起双桨，哦，不对，让我们挽起衣袖和裤脚，来和 PyIntObject 大战一场☺。我们希望在运行时观察 Python 的整数对象系统的变化。这一点，完全可以通过修改 Python 源码来实现。我们修改了 int_print 的行为，让它打印出关于 block_list 和 free_list 的信息，以及小整数缓冲池的信息：

```
static int values[10];
static int refcounts[10];
static int int_print(PyIntObject *v, FILE *fp, int flags)
{
    PyIntObject* intObjectPtr;
    PyIntBlock *p = block_list;
    PyIntBlock *last = NULL;
    int count = 0;
    int i;

    while(p != NULL)
    {
        ++count;
        last = p;
        p = p->next;
    }

    intObjectPtr = last->objects;
    intObjectPtr += N_INTOBJECTS - 1;
    printf(" address @%p\n", v);

    for(i = 0; i < 10; ++i, --intObjectPtr)
    {
        values[i] = intObjectPtr->ob_ival;
        refcounts[i] = intObjectPtr->ob_refcnt;
    }
    printf("  value : ");
    for(i = 0; i < 8; ++i)
    {
        printf("%d\t", values[i]);
    }
    printf("\n");

    printf("  refcnt : ");
    for(i = 0; i < 8; ++i)
    {
        printf("%d\t", refcounts[i]);
    }
    printf("\n");

    printf(" block_list count : %d\n", count);
    printf(" free_list : %p\n", free_list);

    return 0;
}
```

需要特别注意的是，在初始化小整数缓冲池时，对于 block_list 及每个 PyIntBlock 的 objects，都是从后往前开始填充的，所以在初始化完成后，-5 应该在最后一个 PyIntBlock 对象的 objects 内最后一块内存，所以我们需要顺藤摸瓜，一直找到这最后的一块内存，才能观察从-5 到 4 这 10 个小整数。

首先我们创建一个 PyIntObject 对象-9999，从图 2-10 所示的输出信息可以看到，小整数对象很多都被 Python 自身使用多次了。

```
>>> i = -9999
>>> i
address @00C191F0
 value : -5    -4    -3    -2    -1    0     1     2
refcnt : 1     1     2     1     37    112   223   60
block_list count : 6
free_list : 00C191E4
```

图 2-10 整数对象系统内部状态之一

现在的 free_list 指向地址为 00C191E4 的内存，根据上面对 PyIntObject 的分析，那么下一个 PyIntObject 会在这个地址安身立命。那么好，我们接着再建立了两个 PyIntObject 对象，它们的值分别是-12345：

```
>>> a = -12345
>>> a
address @00C191E4
 value : -5    -4    -3    -2    -1    0     1     2
refcnt : 1     1     2     1     37    112   223   60
block_list count : 6
free_list : 00C191FC

>>> b = -12345
>>> b
address @00C191FC
 value : -5    -4    -3    -2    -1    0     1     2
refcnt : 1     1     2     1     37    112   223   60
block_list count : 6
free_list : 00C191D8
```

图 2-11 整数对象系统内部状态之二

从图 2-11 所示的结果中可以看到 a 的地址正是创建 i 后 free_list 所指的地址，而 b 的地址也正是创建 a 后 free_list 所指的地址。虽然 a 和 b 的值都是一样的，但是它们确实是两个完全没有关系的 PyIntObject 对象，这点从地址上看得一清二楚。

现在我们将 b 删除，结果如图 2-12 所示：

```
>>> del b
>>> a
address @00C191E4
 value : -5    -4    -3    -2    -1    0     1     2
refcnt : 1     1     2     1     37    112   223   60
block_list count : 6
free_list : 00C191FC
```

图 2-12 整数对象系统内部状态之三

删除 b 后，free_list 回退到了 a 创建后 free_list 的位置，这一点也跟前面的分析是一致的。

最后我们来看一看对小整数对象的监控，连续两次创建 PyIntObject 对象-5，结果如图 2-13 所示：

```
>>> c1 = -5
>>> c1
address @00AB5948
 value : -5    -4    -3    -2    -1    0     1     2
refcnt : 5     1     2     1     37    112   223   60
block_list count : 6
free_list : 00C191FC

>>> c2 = -5
>>> c2
address @00AB5948
 value : -5    -4    -3    -2    -1    0     1     2
refcnt : 6     1     2     1     37    112   223   60
block_list count : 6
free_list : 00C191FC
```

图 2-13 小整数对象的内部状态

可以看到，两次创建的 PyIntObject 对象 c1 和 c2 的地址都是 00AB5948，这证明它们实际上是同一个对象。同时，我们看到小整数池中-5 的引用计数发生了变化，这证明 c1 和 c2 实际上都是指向这个对象的。此外，free_list 没有发生任何变化。这些都与我们对 PyIntObject 的分析相符。

CHAPTER

第 3 章

# Python 中的字符串对象

在对 PyIntObject 的分析中，我们看到了 Python 中具有不可变长度数据的对象（定长对象）。在 Python 中，还大量存在着另一种对象，即具有可变长度数据的对象（变长对象）。与定长对象不同，对于变长对象而言，对象维护的数据的长度在对象定义时是不知道的。

考虑整数对象 PyIntObject，其维护的数据的长度在对象定义时就已经确定了，它是一个 C 中 long 变量的长度；而可变对象维护的数据的长度只能在对象创建时才能确定，举个例子来说，我们只能在创建一个字符串或一个列表时知道它们所维护的数据的长度，在此之前，对这个信息，我们一无所知。

前一章描述了“可变对象”和“不可变对象”的区别，在变长对象中，实际上也还可分为可变对象和不可变对象。可变对象维护的数据在对象被创建后还能再变化，比如一个 list 被创建后，可以向其中添加元素或删除元素，这些操作都会改变其维护的数据；而不可变对象所维护的数据在对象创建之后就不能再改变了，比如 Python 中的 string 和 tuple，它们都不支持添加或删除的操作。本章我们将研究 Python 变长对象中的不可变对象——字符串对象。

## 3.1 PyStringObject 与 PyString_Type

在 Python 中，PyStringObject 是对字符串对象的实现。PyStringObject 是一个拥有可变长度内存的对象，这一点非常容易理解，因为对于表示“Hi”和“Python”的两个不同的 PyStringObject 对象，其内部所需的保存字符串内容的内存空间显然是不一样的。同时，PyStringObject 对象又是一个不变对象。当创建了一个 PyStringObject 对象之后，该对象内部维护的字符串就不能再被改变了。这一点特性使得 PyStringObject 对象可作为 dict 的键值，但同时也使得一些字符串操作的效率大大降低，比如多个字符

串的连接操作。

PyStringObject 对象的定义如下：

```
[stringobject.h]
typedef struct {
    PyObject_VAR_HEAD
    long ob_shash;
    int ob_sstate;
    char ob_sval[1];
} PyStringObject;
```

在 PyStringObject 的定义中我们看到，在 PyStringObject 的头部实际上是一个 PyObject_VAR_HEAD，其中有一个 ob_size 变量保存着对象中维护的可变长度内存的大小。虽然在 PyStringObject 的定义中，ob_sval 是一个字符的字符数组。但是 ob_sval 实际上是作为一个字符指针指向一段内存的，这段内存保存着这个字符串对象所维护的实际字符串，显然，这段内存不会只是一个字节。而这段内存的实际长度（字节），正是由 ob_size 来维护的，这个机制是 Python 中所有变长对象的实现机制。比如对于 PyStringObject 对象"Python"，ob_size 的值就是 6。

同 C 中的字符串一样，PyStringObject 内部维护的字符串在末尾必须以'\0'结尾，但是由于字符串的实际长度是由 ob_size 维护的，所以 PyStringObject 表示的字符串对象中间是可能出现字符'\0'的，这一点与 C 语言中不同，因为在 C 中，只要遇到了字符'\0'，就认为一个字符串结束了。所以，实际上，ob_sval 指向的是一段长度为 ob_size+1 个字节的内存，而且必须满足 ob_sval[ob_size] == '\0'。

PyStringObject 中的 ob_shash 变量之作用是缓存该对象的 hash 值，这样可以避免每一次都重新计算该字符串对象的 hash 值。如果一个 PyStringObject 对象还没有被计算过 hash 值，那么 ob_shash 的初始值是—1。以后在剖析 Python 中 dict 的时候，我们会看到，这个 hash 值将发挥巨大的作用。在计算一个字符串对象的 hash 值时，采用如下的算法：

```
[stringobject.c]
static long string_hash(PyStringObject *a)
{
    register int len;
    register unsigned char *p;
    register long x;

    if (a->ob_shash != -1)
        return a->ob_shash;
    len = a->ob_size;
    p = (unsigned char *) a->ob_sval;
    x = *p << 7;
    while (--len >= 0)
        x = (1000003*x) ^ *p++;
    x ^= a->ob_size;
```

```
    if (x == -1)
        x = -2;
    a->ob_shash = x;
    return x;
}
```

PyStringObject 对象的 ob_sstate 变量标记了该对象是否已经过 intern 机制的处理，关于 PyStringObject 的 intern 机制，在后面会详细介绍。在 Python 源码中的注释显示，预存字符串的 hash 值和这里的 intern 机制将 Python 虚拟机的执行效率提升了 20%。

下面列出了 PyStringObject 对应的类型对象——PyString_Type:

```
[stringobject.c]
PyTypeObject PyString_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "str",
    sizeof(PyStringObject),
    sizeof(char),
    ……
    (reprfunc)string_repr,          /* tp_repr */
    &string_as_number,              /* tp_as_number */
    &string_as_sequence,            /* tp_as_sequence */
    &string_as_mapping,             /* tp_as_mapping */
    (hashfunc)string_hash,          /* tp_hash */
    0,                  /* tp_call */
    ……
    string_new,                 /* tp_new */
    PyObject_Del,                       /* tp_free */
};
```

可以看到，在 PyStringObject 的类型对象中，tp_itemsize 被设置为 sizeof(char)，即一个字节。对于 Python 中的任何一种变长对象，tp_itemsize 这个域是必须设置的，tp_itemsize 指明了由变长对象保存的元素（item）的单位长度，所谓单位长度即是指一个元素在内存中的长度。这个 tp_itemsize 和 ob_size 共同决定了应该额外申请的内存之总大小是多少。

需要注意的是，我们看到，tp_as_number、tp_as_sequence、tp_as_mapping，三个域都被设置了。这表示 PyStringObject 对数值操作、序列操作和映射操作都支持。

## 3.2 创建 PyStringObject 对象

Python 提供了两条路径，从 C 中原生的字符串创建 PyStringObject 对象。我们先考察一下最一般的 PyString_FromString（见代码清单 3-1）。

代码清单 3-1

```
[stringobject.c]
PyObject* PyString_FromString(const char *str)
```

```
{
    register size_t size;
    register PyStringObject *op;

    //[1]: 判断字符串长度
    size = strlen(str);
    if (size > PY_SSIZE_T_MAX) {
        return NULL;
    }

    //[2]: 处理null string
    if (size == 0 && (op = nullstring) != NULL) {
        return (PyObject *)op;
    }
    //[3]: 处理字符
    if (size == 1 && (op = characters[*str & UCHAR_MAX]) != NULL) {
        return (PyObject *)op;
    }

    /* [4]: 创建新的PyStringObject对象，并初始化 */
    /* Inline PyObject_NewVar */
    op = (PyStringObject *)PyObject_MALLOC(sizeof(PyStringObject) + size);
    PyObject_INIT_VAR(op, &PyString_Type, size);
    op->ob_shash = -1;
    op->ob_sstate = SSTATE_NOT_INTERNED;
    memcpy(op->ob_sval, str, size+1);
    ......
    return (PyObject *) op;
}
```

显然，传给 PyString_FromString 的参数必须是一个指向以 NUL('\0')结尾的字符串的指针。在从一个原生字符串创建 PyStringObject 时，首先在代码清单 3-1 的[1]处检查该字符数组的长度，如果字符数组的长度大于了 PY_SSIZE_T_MAX，那么 Python 将不会创建对应的 PyStringObject 对象。PY_SSIZE_T_MAX 是一个与平台相关的值，在 WIN32 系统下，该值为 2 147 483 647。换算一下，就是 2GB。嗯，这个界限值确实非常庞大了，如果不是变态，几乎没有人会去试图超越这个禁区的☺。

接下来，在代码清单 3-1 的[2]处，检查传入的字符串是不是一个空串。对于空串，Python 并不是每一次都会创建相应的 PyStringObject。Python 运行时有一个 PyStringObject 对象指针 nullstring 专门负责处理空的字符数组。如果第一次在一个空字符串基础上创建 PyStringObject，由于 nullstring 指针被初始化为 NULL，所以 Python 会为这个空字符建立一个 PyStringObject 对象，将这个 PyStringObject 对象通过 intern 机制进行共享，然后将 nullstring 指向这个被共享的对象。如果在以后 Python 检查到需要为一个空字符串创建 PyStringObject 对象，这时 nullstring 已经存在了，那么就直接返回 nullstring 的引用。

如果不是创建空字符串对象，那么接下来需要进行的动作就是申请内存，创建

PyStringObject 对象。可以看到，代码清单 3-1 的[4]处申请的内存除了 PyStringObject 的内存，还有为字符数组内的元素申请的额外内存。然后，将 hash 缓存值设为—1，将 intern 标志设为 SSTATE_NOT_INTERNED。最后将参数 str 指向的字符数组内的字符拷贝到 PyStringObject 所维护的空间中，在拷贝的过程中，将字符数组最后的'\0'字符也拷贝了。加入我们对于字符数组“Python”建立 PyStringObject 对象，那么对象建立完成后在内存中的状态如图 3-1 所示：

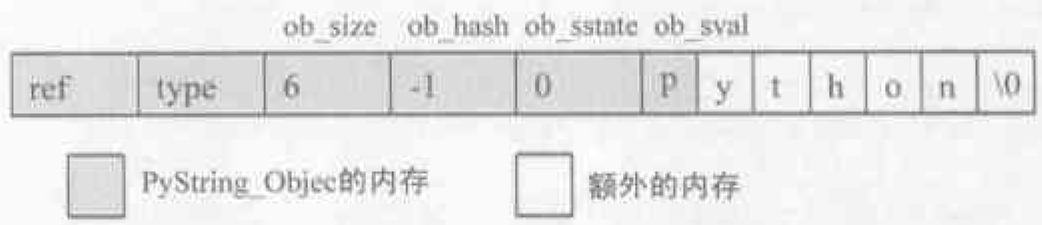



图 3-1　新创建的 PyStringObject 对象的内存布局

在 PyString_FromString 之外，还有一条创建 PyStringObject 对象的途径——PyString_FromStringAndSize：

```
[stringobject.c]
PyObject* PyString_FromStringAndSize(const char *str, int size)
{
    register PyStringObject *op;
    //处理 null string
    if(size == 0 && (op = nullstring) != NULL) {
        return (PyObject *)op;
    }
    //处理字符
    if(size == 1 && str != NULL && (op = characters[*str & UCHAR_MAX]) !=
      NULL)
    {
        return (PyObject *)op;
    }
    //创建新的 PyStringObject 对象，并初始化
    //Inline PyObject_NewVar
    op = (PyStringObject *)PyObject_MALLOC(sizeof(PyStringObject) + size);
    PyObject_INIT_VAR(op, &PyString_Type, size);
    op->ob_shash = -1;
    op->ob_sstate = SSTATE_NOT_INTERNED;
    if (str != NULL)
        memcpy(op->ob_sval, str, size);
    op->ob_sval[size] = '\0';
    ......
    return (PyObject *) op;
}
```

PyString_FromStringAndSize 的操作过程和 PyString_FromString 一般无二，只是有一点，PyString_FromString 传入的参数必须是以 NUL('\0')结尾的字符数组的指针，而 PyString_FromStringAndSize 不会有这样的要求，因为通过传入的 size 参数就可以确定需要拷贝的字符的个数。

## 3.3 字符串对象的 intern 机制

无论是 PyString_FromString 还是 PyString_FromStringAndSize，我们都注意到，当字符数组的长度为 0 或 1 时，需要进行一个特别的动作：PyString_InternInPlace。这就是前面所提到的 intern 机制。

```
[stringobject.c]
PyObject* PyString_FromString(const char *str)
{
    register size_t size;
    register PyStringObject *op;

    ……//创建 PyStringObject 对象

    // intern（共享）长度较短的 PyStringObject 对象
    if (size == 0) {
        PyObject *t = (PyObject *)op;
        PyString_InternInPlace(&t);
        op = (PyStringObject *)t;
        nullstring = op;
    } else if (size == 1) {
        PyObject *t = (PyObject *)op;
        PyString_InternInPlace(&t);
        op = (PyStringObject *)t;
        characters[*str & UCHAR_MAX] = op;
    }
    return (PyObject *) op;
}
```

PyStringObject 对象的 intern 机制之目的是：对于被 intern 之后的字符串，比如“Ruby”，在整个 Python 的运行期间，系统中都只有唯一的一个与字符串“Ruby”对应的 PyStringObject 对象。这样当判断两个 PyStringObject 对象是否相同时，如果它们都被 intern 了，那么只需要简单地检查它们对应的 PyObject*是否相同即可。这个机制既节省了空间，又简化了对 PyStringObject 对象的比较，嗯，可谓是一箭双雕哇。通过下面展示的一段 Python 代码，我们来考察一下 intern 机制的必要性：

```
a = "Python"
b = "Python"
print a, b
```

首先，我们创建了一个 PyStringObject 对象 a，其表示的字符串是“Python”，随后，我们再一次为字符串“Python”创建一个 PyStringObject 对象。通常情况下，Python 会为我们重新申请内存，创建一个新的 PyStringObject 对象 b，a 与 b 是完全不同的两个对象，尽管其内部维护的字符数组是完全相同的。

这就带来了一个问题，假如我们在程序中创建了 100 个“Python”的 PyStringObject 对象呢？显而易见，这样会大量地浪费珍贵的内存。因此 Python 为 PyStringObject 对

象引入了 intern 机制。在上面的例子中，如果对于 a 应用了 intern 机制，那么之后要创建 b 的时候，Python 会首先在系统中记录的已经被 intern 机制处理了的 PyStringObject 对象中查找，如果发现该字符数组对应的 PyStringObject 对象已经存在了，那么就将该对象的引用返回，而不再重新创建一个 PyStringObject 对象。PyString_InternInPlace 正是负责完成对一个对象进行 intern 操作的函数（见代码清单 3-2）。

代码清单 3-2

```
[stringobjec.c]
void PyString_InternInPlace(PyObject **p)
{
    register PyStringObject *s = (PyStringObject *)(*p);
    PyObject *t;
    //对 PyStringObject 进行类型和状态检查
    if (!PyString_CheckExact(s))
        return;
    if (PyString_CHECK_INTERNED(s))
        return;
    //创建记录经 intern 机制处理后的 PyStringObject 的 dict
    if (interned == NULL) {
        interned = PyDict_New();
    }
    //[1] : 检查 PyStringObject 对象 s 是否存在对应的 intern 后的 PyStringObject 对象
    t = PyDict_GetItem(interned, (PyObject *)s);
    if (t) {
        //注意这里对引用计数的调整
        Py_INCREF(t);
        Py_DECREF(*p);
        *p = t;
        return;
    }
    //[2] : 在 interned 中记录检查 PyStringObject 对象 s
    PyDict_SetItem(interned, (PyObject *)s, (PyObject *)s);
    //[3] : 注意这里对引用计数的调整
    s->ob_refcnt -= 2;
    //[4] : 调整 s 中的 intern 状态标志
    PyString_CHECK_INTERNED(s) = SSTATE_INTERNED_MORTAL;
}
```

PyString_InternInPlace 首先会进行一系列的检查，其中包括两项检查内容：

- 检查传入的对象是否是一个 PyStringObject 对象，intern 机制只能应用在 PyStringObject 对象上，甚至对于它的派生类对象系统都不会应用 intern 机制。
- 检查传入的 PyStringObject 对象是否已经被 intern 机制处理过了，Python 不会对同一个 PyStringObject 对象进行一次以上的 Intern 操作。

从代码中我们可以清楚地看到，intern 机制的核心在于 interned 这个东西，那么这个东西是个什么东西呢？interned 在 stringobject.c 中被定义为：static PyObject *interned。

从 stringobject.c 中的定义我们完全不知道 interned 是个什么东西，然而在这里我们看到，interned 实际上指向的是 PyDict_New 创建的一个对象。而 PyDict_New 实际上创建了一个 PyDictObject 对象，也就是我们在 Python 中经常用到的“dict”啦。这个对象将在后面的章节中被详细地剖析。现在，对于一个 PyDictObject 对象，我们完全可以看作是 C++中的 map，即 map<PyObject*, PyObject*>。

现在一切都清楚了，所谓的 intern 机制的关键，就是在系统中有一个（key，value）映射关系的集合，集合的名称叫做 interned。在这个集合中，记录着被 intern 机制处理过的 PyStringObject 对象。当对一个 PyStringObject 对象 a 应用 intern 机制时，首先会在 interned 这个 dict 中检查是否有满足以下条件的对象 b：b 中维护的原生字符串与 a 相同。如果确实存在对象 b，那么指向 a 的 PyObject 指针将会指向 b，而 a 的引用计数减 1，这样，其实 a 只是一个被临时创建的对象。如果 interned 中还不存在这样的 b，那么就在代码清单 3-2 的[2]处将 a 记录到 interned 中。

图 3-2 展示了如果 interned 中存在这样的对象 b，在对 a 进行 intern 操作时，原本指向 a 的 PyObject*指针的变化：

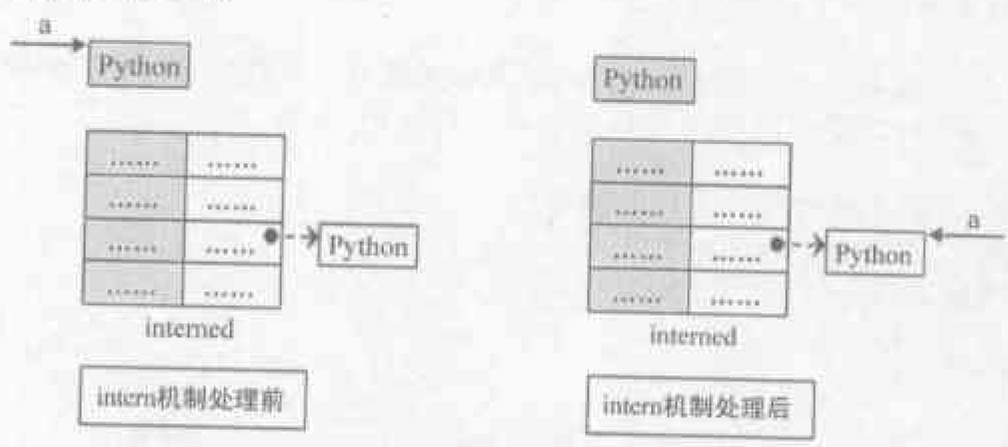



图 3-2 intern 机制示意图

对于被 intern 机制处理了的 PyStringObject 对象，Python 采用了特殊的引用计数机制。在将一个 PyStringObject 对象 a 的 PyObject 指针作为 key 和 value 添加到 interned 中时，PyDictObject 对象会通过这两个指针对 a 的引用计数进行两次加 1 的操作。但是 Python 的设计者规定在 interned 中 a 的指针不能被视为对象 a 的有效引用，因为如果是有效引用的话，那么 a 的引用计数在 Python 结束之前永远都不可能为 0，因为 interned 中至少有两个指针引用了 a，那么删除 a 就永远不可能了，这显然是没有道理的。

因此，interned 中的指针不能作为 a 的有效引用。这也就是在代码清单 3-2 的[3]处会将引用计数减 2 的原因。在 A 的引用计数在某个时刻减为 0 之后，系统将会销毁对象 a，那么我们可以预期，在销毁 a 的同时，会在 interned 中删除指向 a 的指针，显然，这一点在下面列出的 string_dealloc 代码中得到了验证：

```
[stringobject.c]
static void string_dealloc(PyObject *op)
```

```
{
    switch (PyString_CHECK_INTERNED(op)) {
        case SSTATE_NOT_INTERNED:
            break;

        case SSTATE_INTERNED_MORTAL:
            /* revive dead object temporarily for DelItem */
            op->ob_refcnt = 3;
            if (PyDict_DelItem(interned, op) != 0)
                Py_FatalError(
                    "deletion of interned string failed");
            break;

        case SSTATE_INTERNED_IMMORTAL:
            Py_FatalError("Immortal interned string died.");

        default:
            Py_FatalError("Inconsistent interned string state.");
    }
    op->ob_type->tp_free(op);
}
```

前面提到，Python 在创建一个字符串时，会首先在 interned 中检查是否已经有该字符串对应的 PyStringObject 对象了，如果有，则不用创建新的，这样可以节省内存空间。事到如今，我必须承认，我说谎了☺。

节省内存空间是没错的，可是 Python 并不是在创建 PyStringObject 时就通过 interned 实现了节省空间的目的。事实上，从 PyString_FromString 中可以看到，无论如何，一个合法的 PyStringObject 对象是会被创建的，同样，我们可以注意到，PyString_InternInPlace 也只对 PyStringObject 起作用。事实正是如此，Python 始终会为字符串 s 创建 PyStringObject 对象，尽管 s 中维护的原生字符数组在 interned 中已经有一个与之对应的 PyStringObject 对象了。而 intern 机制是在 s 被创建后才起作用的，通常 Python 在运行时创建了一个 PyStringObject 对象 temp 后，基本上都会调用 PyString_InternInPlace 对 temp 进行处理，intern 机制会减少 temp 的引用计数，temp 对象会由于引用计数减为 0 而被销毁，它只是作为一个临时对象昙花一现地在内存中闪现，然后湮灭。

现在读者可能有一个疑问了，是否可以直接在 C 的原生字符串上做 intern 的动作，而不需要再创建这样一个临时对象呢？事实上，Python 确实提供了一个以 char*为参数的 intern 机制相关函数，但是你会相当失望，嗯，因为它基本上是换汤不换药的：

```
[stringobject.c]
PyObject* PyString_InternFromString(const char *cp)
{
    PyObject *s = PyString_FromString(cp);
    PyString_InternInPlace(&s);
    return s;
}
```

临时对象照样被创建出来，实际上，仔细一想，就会发现在 Python 中，必须创建这样一个临时的 PyStringObject 对象来完成 intern 操作。为什么呢？答案就在 PyDictObject 对象 interned 中，因为 PyDictObject 必须以 PyObject*指针作为键。

关于 PyStringObject 对象的 intern 机制，还有一点需要注意。实际上，被 intern 机制处理后的 PyStringObject 对象分为两类，一类处于 SSTATE_INTERNED_IMMORTAL 状态，而另一类则处于 SSTATE_INTERNED_MORTAL 状态，这两种状态的区别在 string_dealloc 中可以清晰地看到，显然，SSTATE_INTERNED_IMMORTAL 状态的 PyStringObject 对象是永远不会被销毁的，它将与 Python 虚拟机同年同月同日死。

PyString_InternInPlace 只能创建 SSTATE_INTERNED_MORTAL 状态的 PyStringObject 对象，如果想创建 SSTATE_INTERNED_IMMORTAL 状态的对象，必须要通过另外的接口，在调用了 PyString_InternInPlace 后，强制改变 PyStringObject 的 intern 状态。

```
[stringobject.c]
void PyString_InternImmortal(PyObject **p)
{
    PyString_InternInPlace(p);
    if (PyString_CHECK_INTERNED(*p) != SSTATE_INTERNED_IMMORTAL) {
        PyString_CHECK_INTERNED(*p) = SSTATE_INTERNED_IMMORTAL;
        Py_INCREF(*p);
    }
}
```

## 3.4 字符缓冲池

在上一章对整数对象的剖析中我们看到，Python 为小整数对象贴心地准备了专有的缓冲池。在“尊老爱幼”这一点上，Python 是做得相当好的，类似于小整数对象的对象池，Python 的设计者为 PyStringObject 中的一个字节的字符对应的 PyStringObject 对象也设计了这样一个对象池 characters：

```
static PyStringObject *characters[UCHAR_MAX + 1];
```

其中的 UCHAR_MAX 是在系统头文件中定义的常量，这也是一个平台相关的常量，在 Win32 平台下：

```
#define UCHAR_MAX     0xff      /* maximum unsigned char value */
```

在 Python 的整数对象体系中，小整数的缓冲池是在 Python 初始化时被创建的，而字符串对象体系中的字符缓冲池则是以静态变量的形式存在着的。在 Python 初始化完成之后，缓冲池中的所有 PyStringObject 指针都为空。

当我们在创建一个 PyStringObject 对象时，无论是通过调用 PyString_FromString 还是通过调用 PyString_FromStringAndSize，如果字符串实际上是一个字符，则会进行

如下的操作：

```
[stringobject.c]
PyObject* PyString_FromStringAndSize(const char *str, int size)
{
    ......
    else if (size == 1 && str != NULL)
    {
        PyObject *t = (PyObject *)op;
        PyString_InternInPlace(&t);
        op = (PyStringObject *)t;
        characters[*str & UCHAR_MAX] = op;
        Py_INCREF(op);
    }
    return (PyObject *) op;
}
```

先对所创建的字符串（字符）对象进行 intern 操作，再将 intern 的结果缓存到字符缓冲池 characters 中。图 3-3 演示了缓存一个字符对应的 PyStringObject 对象的过程。

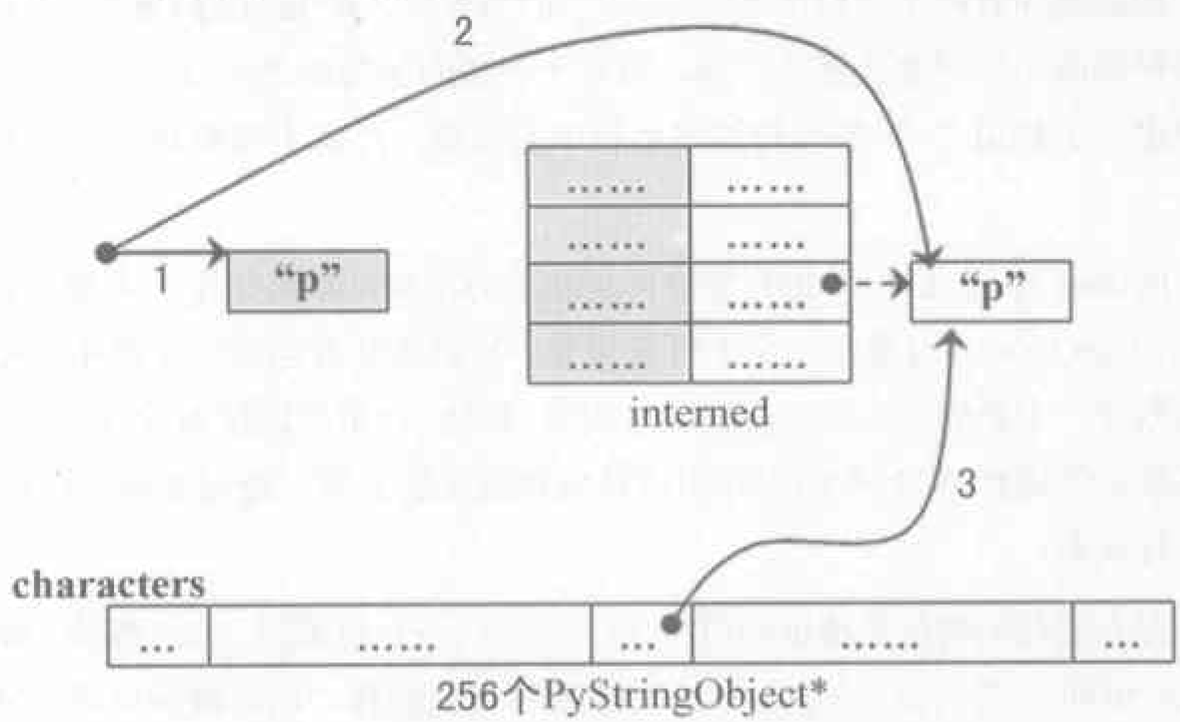



图 3-3 创建字符对应的 PyStringObject 对象

3 条带有标号的曲线既代表指针，又代表进行操作的顺序：

（1） 创建 PyStringObject 对象<string P>；

（2） 对对象<string P>进行 intern 操作；

（3） 将对象<string P>缓存至字符缓冲池中。

同样，在创建 PyStringObject 时，会首先检查所要创建的是否是一个字符对象，然后检查字符缓冲池中是否已经有了这个字符的字符对象的缓冲，如果有，则直接返回这个缓冲的对象即可：

```
[stringobject.c]
PyObject* PyString_FromStringAndSize(const char *str, int size)
{
```

```
    register PyStringObject *op;
    ……
    if(size == 1 && str != NULL && (op = characters[*str & UCHAR_MAX]) != NULL)
    {
        return (PyObject *)op;
    }
    ……
}
```

## 3.5 PyStringObject 效率相关问题

关于 PyStringObject，有一个地球人都知道的严重影响 Python 程序执行效率的问题，有一种说法，绝大部分执行效率特别低下的 Python 程序都是由于没有注意这个问题所致。下面我们就来看看这个在 Python 中举足轻重的问题——字符串连接。

假如现在有两个字符串“Python”和“Ruby”，在 Java 或 C#中，都可以通过使用“+”操作符将两个字符串连接在一起，得到一个新的字符串“PythonRuby”。当然，Python 中同样提供了利用“+”操作符连接字符串的功能，然而不幸的是，这样的做法正是万恶之源。

Python 中通过“+”进行字符串连接的方法效率极其低下，其根源在于 Python 中的 PyStringObject 对象是一个不可变对象。这就意味着当进行字符串连接时，实际上是必须要创建一个新的 PyStringObject 对象。这样，如果要连接 N 个 PyStringObject 对象，那么就必须进行 N-1 次的内存申请及内存搬运的工作。毫无疑问，这将严重影响 Python 的执行效率。

官方推荐的做法是通过利用 PyStringObject 对象的 join 操作来对存储在 list 或 tuple 中的一组 PyStringObject 对象进行连接操作，这种做法只需要分配一次内存，执行效率将大大提高。

下面我们通过考察源码来更细致地了解这一问题。

通过“+”操作符对字符串进行连接时，会调用 string_concat 函数：

```
[stringobject.c]
static PyObject* string_concat(register PyStringObject *a, register
  PyObject *bb)
{
    register unsigned int size;
    register PyStringObject *op;
    #define b ((PyStringObject *)bb)
    ……
    //计算字符串连接后的长度size
    size = a->ob_size + b->ob_size;
```

```
    /* Inline PyObject_NewVar */
    //创建新的 PyStringObject 对象，其维护的用于存储字符的内存长度为 size
    op = (PyStringObject *)PyObject_MALLOC(sizeof(PyStringObject) + size);
    PyObject_INIT_VAR(op, &PyString_Type, size);
    op->ob_shash = -1;
    op->ob_sstate = SSTATE_NOT_INTERNED;

    //将 a 和 b 中的字符拷贝到新创建的 PyStringObject 中
    memcpy(op->ob_sval, a->ob_sval, (int) a->ob_size);
    memcpy(op->ob_sval + a->ob_size, b->ob_sval, (int) b->ob_size);
    op->ob_sval[size] = '\0';
    return (PyObject *) op;
    #undef b
}
```

对于任意两个 PyStringObject 对象的连接，就会进行一次内存申请的动作。而如果利用 PyStringObject 对象的 join 操作，则会进行如下的动作（假设是对 list 中的 PyStringObject 对象进行连接）：

```
[stringobject.c]
static PyObject* string_join(PyStringObject *self, PyObject *orig)
{
    char *sep = PyString_AS_STRING(self);
    //假设调用"abc".join(list)，那么 self 就是 "abc" 对应的 PyStringObject 对象
    //所以 seplen 中存储这 "abc" 的长度
    const int seplen = PyString_GET_SIZE(self);
    PyObject *res = NULL;
    char *p;
    int seqlen = 0;
    size_t sz = 0;
    int i;
    PyObject *seq, *item;
    ……//获得 list 中 PyStringObject 对象的个数，保存在 seqlen 中

    //遍历 list 中每一个字符串，累加获得连接 list 中所有字符串后的长度
    for (i = 0; i < seqlen; i++)
    {
        //seq 为 Python 中的 list 对象，这里获得其中第 i 个字符串
        item = PySequence_Fast_GET_ITEM(seq, i);
        sz += PyString_GET_SIZE(item);
        if (i != 0)
            sz += seplen;
    }
    //创建长度为 sz 的 PyStringObject 对象
    res = PyString_FromStringAndSize((char*)NULL, (int)sz);
    //将 list 中的字符串拷贝到新创建的 PyStringObject 对象中
    p = PyString_AS_STRING(res);
    for (i = 0; i < seqlen; ++i)
    {
        size_t n;
        item = PySequence_Fast_GET_ITEM(seq, i);
        n = PyString_GET_SIZE(item);
        memcpy(p, PyString_AS_STRING(item), n);
        p += n;
```

```
        if (i < seqlen - 1)
        {
            memcpy(p, sep, seplen);
            p += seplen;
        }
    }
    return res;
}
```

执行 join 操作时，会首先统计出在 list 中一共有多少个 PyStringObject 对象，并统计这些 PyStringObject 对象所维护的字符串一共有多长，然后申请内存，将 list 中所有的 PyStringObject 对象维护的字符串都拷贝到新开辟的内存空间中。注意，这里只进行了一次内存空间的申请，就完成了 N 个 PyStringObject 对象的连接操作。相比于"+"操作符，待连接的 PyStringObject 对象越多，效率的提升也越明显。

通过在 string_concat 和 string_join 中添加输出代码，我们可以在图 3-4 中形象地看到两种连接字符串的方法的区别：

```
>>> 's1' + 's2' + 's3'
call string_concat
call string_concat
's1s2s3'
>>>
>>> ' '.join(['s1', 's2', 's3'])
call string_join
's1 s2 s3'
>>>
```

图 3-4 concat 与 join 的区别

## 3.6 Hack PyStringObject

同上一章的 Hack PyIntObject 一样，在这一节，我们将对 PyStringObject 对象在运行时的行为进行两项观察。

首先，观察 intern 机制，在 Python Interactive 环境中，创建一个 PyStringObject 后，就会对这个 PyStringObject 对象进行 intern 操作，所以我们预期内容相同的 PyStringObject 对象在 intern 后应该是同一个对象，图 3-5 是观察的结果：

```
>>> c1 = 'x'                    >>> s1 = "python"
>>> len(c1)                     >>> len(s1)
address : 0xB8C420              address : 0xB91060
3                               2
>>> c2 = 'x'                    >>> s2 = "python"
>>> len(c2)                     >>> len(s2)
address : 0xB8C420              address : 0xB91060
4                               3
>>>                             ...
```

图 3-5 intern 机制的观察结果

通过在 string_length 中添加打印地址和引用计数的代码，我们可以在 Python 运行期间获得每一个 PyStringObject 对象的地址及其引用计数(在 address 下一行输出的不是字符串的长度信息，我们已经将之更换为引用计数信息)。从观察结果中可以看到，无论是对于一般的字符串，还是对于单个字符，intern 机制最终都会使不同的 PyStringObject* 指针指向相同的对象。

然后，我们观察 Python 中进行缓冲处理的字符对象，同样是通过在 string_length 中添加代码，打印出缓冲池中从 a 到 e 的字符对象的引用计数信息。需要注意的是，为了避免执行 len(…) 对引用计数的影响，我们并不会对 a 到 e 的字符对象调用 len 操作，而是对另外的 PyStringObject 对象调用 len 操作：

```
static void ShowCharacter()
{
  char chA = 'a';
  PyStringObject** posA = characters+(unsigned short)chA;
  int i;
  char values[5];
  int refcnts[5];
  for(i = 0; i < 5; ++i)
  {
      PyStringObject* strObj = posA[i];
      values[i] = strObj->ob_sval[0];
      refcnts[i] = strObj->ob_refcnt;
  }

  printf(" value : ");
  for(i = 0; i < 5; ++i)
  {
      printf("%c\t", values[i]);
  }
  printf("\nrefcnt : ");
  for(i = 0; i < 5; ++i)
  {
      printf("%d\t", refcnts[i]);
  }
  printf("\n");
}
```

图 3-6 展示了观察的结果，可以看到，在创建字符对象时，Python 确实只是使用了缓冲池里的对象，而没有创建新的对象：

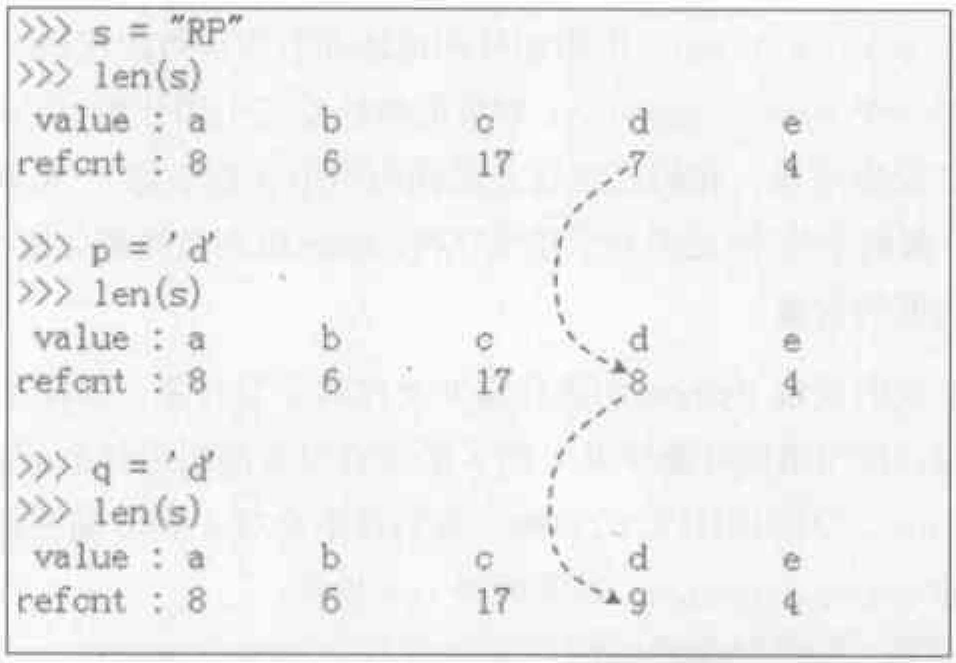



图 3-6 Python 内部的字符缓冲池

CHAPTER

第 4 章

# Python 中的 List 对象

元素的一个群是一个非常重要的抽象概念，我们可以将符合某一特性的一堆元素聚集为一个群，当然，还要可以向群中添加或删除元素。这样的群的概念对于编程语言十分重要，C 语言就内建了数组的概念，随着编程语言的发展，现在所有的编程语言都会在语言中或标准库中实现这样的群。而且群的概念还进一步地细分为多种实现方式，比如 map、vector 等。每一种实现都为某种目的的元素聚集或元素访问提供了极大的方便。

PyListObject 是 Python 提供的对列表的抽象，如果你熟悉 C++，那么很可能会条件反射地将 PyListObject 与 C++中的 list 对应起来（至少它们名字是相似的）。然而实际上，Python 中的列表和 C++的 STL 中 list 是大相径庭，相反，它与 STL 中的 vector 却更为神似。

## 4.1 PyListObject 对象

PyListObject 对象可以有效地支持对元素的插入、添加、删除等操作，在 Python 的列表中，无一例外地存放的都是 PyObject*指针。所以实际上，我们可以这样看待 Python 中的 PyListObject: vector<PyObject*>。

很显然，PyListObject 一定是一个变长对象，因为不同的 list 中存储的元素个数会是不同的。但是，和 PyStringObject 不同的是，PyListObject 对象还支持插入删除等操作，可以在运行时动态地调整其所维护的内存和元素，所以，它还是一个可变对象。

先来看一看 PyListObject 的定义：

```
[listobject.h]
typedef struct {
    PyObject_VAR_HEAD
    //ob_item 为指向元素列表的指针，实际上，Python 中的 list[0]就是 ob_item[0]
```

```
    PyObject **ob_item;
    int allocated;
} PyListObject;
```

如我们所想，在 PyListObject 的头，赫然是一个 PyObject_VAR_HEAD，随后是一个类型为 PyObject**的 ob_item，这个指针和紧接着的 allocated 数值正是维护元素列表（也就是 PyObject*列表）的关键。指针指向了元素列表所在的内存块的首地址，而 allocated 中则维护了当前列表中的可容纳的元素的总数。

我们知道在 PyObject_VAR_HEAD 中，有一个 ob_size，而在 PyListObject 的最后，又有一个 allocated，那么这两个变量之间的关系是什么呢？

其实，ob_size 和 allocated 都和 PyListObject 对象的内存管理有关，PyListObject 所采用的内存管理策略和 C++中 vector 采取的内存管理策略是一样的。它并不是存了多少东西就申请对应大小的内存，这样的内存管理策略显然是低效的，因此我们有理由相信，用户选用列表正是为了频繁地插入或删除元素。所以，在每一次需要申请内存的时候，PyListObject 总会申请一大块内存，这时申请的总内存的大小记录在 allocated 中，而其中实际被使用了的内存的数量则记录在了 ob_size 中。假如有一个能容纳 10 个元素的 PyListObject 对象已经装入了 5 个元素，那对于这个 PyListObject 对象，其 ob_size 为 5，而 allocated 则为 10。

所以，我们不难得到，对于一个 PyListObject 对象，一定存在以下的关系：

```
0 <= ob_size <= allocated
len(list) == ob_size
ob_item == NULL 意味着 ob_size == allocated == 0
```

这里 ob_size 和 allocated 的关系就像 C++的 vector 中 size 和 capacity 的关系一样。从这里我们实际上已然可以窥见 PyListObject 管理元素的策略了。

## 4.2 PyListObject 对象的创建与维护

### 4.2.1 创建对象

为了创建一个列表，Python 只提供了唯一的一条途径——PyList_New。这个函数接受一个 size 参数，从而允许我们可以在创建一个 PyListObject 对象的同时指定该列表初始的元素个数。需要注意，这里仅仅指定了元素的个数，并没有指定元素是什么。我们来看一看 Python 创建 PyListObject 对象的过程（见代码清单 4-1）。

**代码清单 4-1**

```
[listobject.c]
PyObject* PyList_New(int size)
```

```
{
    PyListObject *op;
    size_t nbytes;

    //[1] : 内存数量计算，溢出检查
    nbytes = size * sizeof(PyObject *);
    if (nbytes / sizeof(PyObject *) != (size_t)size)
        return PyErr_NoMemory();

    //[2] : 为 PyListObject 对象申请空间
    if (num_free_lists) {
        //缓冲池可用
        num_free_lists--;
        op = free_lists[num_free_lists];
        _Py_NewReference((PyObject *)op);
    } else {
        //缓冲池不可用
        op = PyObject_GC_New(PyListObject, &PyList_Type);
    }
    //[3] : 为 PyListObject 对象中维护的元素列表申请空间
    if (size <= 0)
        op->ob_item = NULL;
    else {
        op->ob_item = (PyObject **) PyMem_MALLOC(nbytes);
        memset(op->ob_item, 0, nbytes);
    }
    op->ob_size = size;
    op->allocated = size;
    return (PyObject *) op;
}
```

首先，Python 在代码清单 4-1 的[1]处会计算需要使用的内存总量，因为 PyList_New 指定的仅仅是元素的个数，而不是元素实际将占用的内存空间。在这里，Python 会检查指定的元素个数是否会大到使所需内存数量产生溢出的程度，如果会产生溢出，那么 Python 将不会进行任何动作。

接下来，就是 Python 对列表对象的创建动作。我们可以清晰地看到，Python 中的列表对象实际上是分为两部分的，一是 PyListObject 对象本身，二则是 PyListObject 对象维护的元素列表。这是两块分离的内存，它们通过 ob_item 建立了联系。在本章的描述中，我们并不严格地区分 PyListObject 对象和其维护的元素列表之间的差异，有时候我们会以 PyListObject 对象来代表 Python 中元素列表，有时我们则会清晰地指出"PyList-Object 对象中维护的列表对象"，根据上下文环境，读者可以很容易地加以区分和理解。

在代码清单 4-1 的[2]创建新的 PyListObject 对象时，我们看到了现在已经非常熟悉的 Python 对象级的缓冲池技术。在创建 PyListObject 对象时，会首先检查缓冲池 free_lists 中是否有可用的对象，如果有，则直接使用这个可用对象；如果缓冲池中所有对象都不可用，则会通过 PyObject_GC_New 在系统堆中申请内存，创建新的 PyList-

Object 对象。实际上，PyObject_GC_New 除了申请内存之外，还会为 Python 中的自动垃圾收集机制做一些准备工作，不过在这里，我们还不打算深入到 Python 的垃圾收集机制中，对垃圾收集机制的考察会作为单独的一章在后面讲解，这里，我们只需要将 PyObject_GC_New 想象成 malloc 即可。在 Python 2.5 中，默认情况下，free_lists 中最多会维护 80 个 PyListObject 对象。

```
[listobject.c]
    #define MAXFREELISTS 80
    static PyListObject *free_lists[MAXFREELISTS];
    static int num_free_lists = 0;
```

当 Python 创建了新的 PyListObject 对象之后，在代码清单 4-1 的[3]处，会立即根据调用 PyList_New 时传递的 size 参数创建 PyListObject 对象所维护的元素列表。在这里创建的 PyListObject*列表，其中的每一个元素都会被初始化为 NULL 值。

完成了 PyListObject 对象及其维护的列表的创建之后，Python 会调整该 PyListObject 对象，用于维护元素列表中元素数量的 ob_size 和 allocated 两个变量。

细心的读者一定注意到了，在代码清单 4-1 的[2]处提及的 PyListObject 对象缓冲池，实际有一个很奇特的地方。我们看到，在 free_lists 中缓存的只是 PyListObject*，那么这个缓冲池里的 PyListObject*究竟指向什么地方呢？换句话说，这些 PyListObject*指向的 PyListObject 对象是在何时何地被创建的呢？

列位看官，花开两朵，各表一枝。我们先把这个问题放一放，看一看在 Python 开始运行时，第一个 PyListObject 对象被创建时的情形。嗯，这有点像上帝创世纪，挺有趣的☺。

## 4.2.2 设置元素

在第一个 PyListObject 创建的时候，这时的 num_free_lists 是 0，所以在代码清单 4-1 的[2]处会绕过对象缓冲池，转而调用 PyObject_GC_New 在系统堆上创建一个新的 PyListObject 对象，假设我们创建的 PyListObject 是包含 6 个元素的 PyListObject，也就是通过 PyList_New(6)来创建 PyListObject 对象，在 PyList_New 完成之后，这混沌初开的第一个 PyListObject 对象的情形可以从图 4-1 中看到。

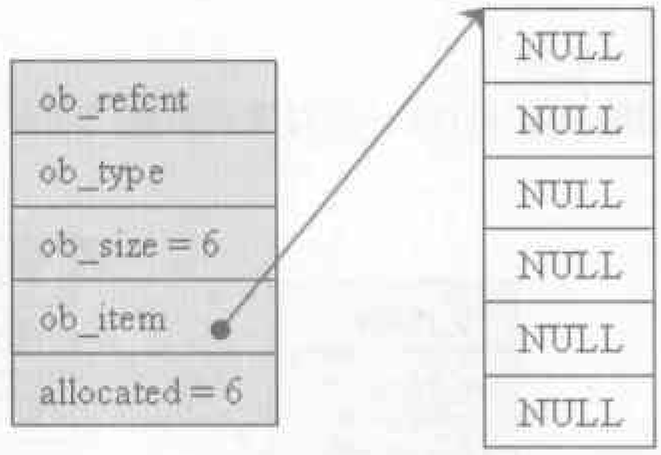



图 4-1 新创建的 PyListObject 对象

需要注意的是，当我们在 Python 的交互式环境或者.py 源文件中创建一个 list 时，内存中的 PyListObject 对象中元素列表中的元素不可能是 NULL。这里我们只是为了演示元素列表的变化，所以不必在意元素是否可以为 NULL。

一个什么东西都没有的 list 当然是很无趣的，我们来尝试向里边添加一点东西，把一个整数对象 100 放到第 4 个位置上去，用 Python 的行话来说就是 list[3] = 100（见代码清单 4-2）。

**代码清单 4-2**

```
[listobject.c]
int PyList_SetItem(register PyObject *op, register int i, register PyObject
                    *newitem)
{
    register PyObject *olditem;
    register PyObject **p;
    //[1] : 索引检查
    if (i < 0 || i >= ((PyListObject *)op) -> ob_size) {
        PyErr_SetString(PyExc_IndexError, "list assignment index out of
          range");
        return -1;
    }
    //[2] : 设置元素
    p = ((PyListObject *)op) -> ob_item + i;
    olditem = *p;
    *p = newitem;
    Py_XDECREF(olditem);
    return 0;
}
```

当我们在 Python 中运行 list[3] = 100 时，在 Python 内部，就是调用 PyList_SetItem 来完成这个动作。首先 Python 会进行类型检查，在这里我们省略了。随后，在代码清单 4-2 的[1]处，会进行索引的有效性检查。当类型检查和索引有效性检查都顺利通过之后，Python 在代码清单 4-2 的[2]处将待加入的 PyObject*指针放到指定的位置，然后调整引用计数，将这个位置原来存放的对象的引用计数减 1。这里的 olditem 很可能会是 NULL，比如向一个新创建的 PyListObject 对象加入元素，就会碰到这样的情况，所以这里必须

使用 Py_XDECREF。

好了，现在我们的 PyListObject 对象再不是当年那个一穷二白的可怜虫了，其情形如图 4-2 所示：

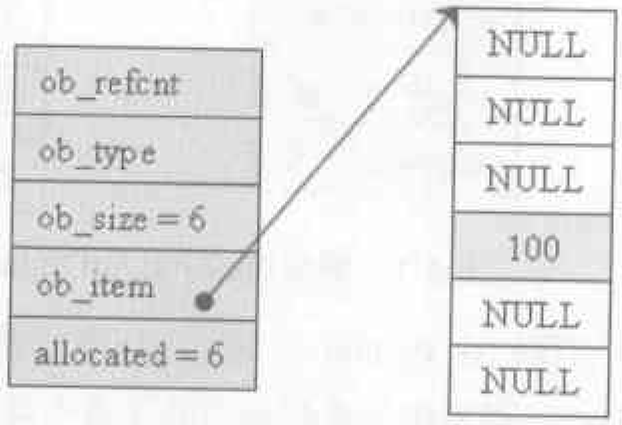



图 4-2 设置元素后的 PyListObject 对象

## 4.2.3 插入元素

设置元素和插入元素的动作是不同的，设置元素不会导致 ob_item 指向的内存发生变化，而插入元素的动作则有可能使得 ob_item 指向的内存发生变化。图 4-3 中显示了设置元素和插入元素的区别：

```
>>> lst = [0, 0, 0, 0, 0, 0]
>>> lst[3] = 100
>>> lst
[0, 0, 0, 100, 0, 0]
>>> lst.insert(3, 99)
>>> lst
[0, 0, 0, 99, 100, 0, 0]
```

图 4-3 设置元素与插入元素

其中的 lst[3] = 100 正是我们在上一节讨论过的设置元素的动作，而 lst.insert(3, 99) 则是插入元素的动作，从图 4-3 的结果可以看到，这个插入动作确实导致了元素列表的内存的变化。接下来会深入地剖析插入元素的动作是如何导致元素列表的内存发生变化的（见代码清单 4-3），我们就以图 4-3 所示的在 100 之前插入 99 为例。99 确实是在 100 之前的，这个地球人都知道☺。

**代码清单 4-3**

```
[listobject.c]
int PyList_Insert(PyObject *op, Py_ssize_t where, PyObject *newitem)
{
    ....../类型检查
    return ins1((PyListObject *)op, where, newitem);
}

static int ins1(PyListObject *self, Py_ssize_t where, PyObject *v)
```

```
{
    Py_ssize_t i, n = self->ob_size;
    PyObject **items;
    ......
    //[1]：调整列表容量
    if (list_resize(self, n+1) == -1)
        return -1;
    //[2]：确定插入点
    if (where < 0) {
        where += n;
        if (where < 0)
            where = 0;
    }
    if (where > n)
        where = n;
    //[3]：插入元素
    items = self->ob_item;
    for (i = n; --i >= where; )
        items[i+1] = items[i];
    Py_INCREF(v);
    items[where] = v;
    return 0;
}
```

Python 内部通过调用 PyList_Insert 来完成元素的插入动作，而 PyList_Insert 实际上是调用了 Python 内部的 ins1。在 ins1 中，为了完成元素的插入工作，必须首先保证一个条件得到满足，那就是 PyListObject 对象必须有足够的内存来容纳我们期望插入的元素。Python 通过在代码清单 4-3 的[1]处调用了 list_resize 函数来保证该条件一定能成立。仅仅从函数名我们就可以想象，这个函数一定是改变了 PyListObject 所维护的 PyObject*列表的大小。

```
[listobject.c]
static int list_resize(PyListObject *self, int newsize)
{
    PyObject **items;
    size_t new_allocated;
    int allocated = self->allocated;

    //不需要重新申请内存
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        self->ob_size = newsize;
        return 0;
    }

    //计算重新申请的内存大小
    new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6) + newsize;
    if (newsize == 0)
        new_allocated = 0;
    //扩展列表
    items = self->ob_item;
    PyMem_RESIZE(items, PyObject *, new_allocated); //最终调用 C 中的 realloc
    self->ob_item = items;
```

```
    self->ob_size = newsize;
    self->allocated = new_allocated;
    return 0;
}
```

在调整 PyListObject 对象所维护的列表的内存时，Python 分两种情况处理：

- newsize < allocated && newsize > allocated/2 ：简单调整 ob_size 值；
- 其他情况，调用 realloc，重新分配空间。

可以看出，Python 对内存可谓是殚精竭虑了，甚至在第 2 种情况中，当 newsize 与 allocated 的关系满足 newsize < allcated/2 的时候，Python 甚至还会通过 realloc 来收缩列表的内存空间，真是恨不得把一个字节掰成两个字节来用。

将 PyListObject 的空间调整之后，函数 ins1 在实际插入元素之前还需要在代码清单 4-3 的[2]处首先确定元素的插入点。Python 的 list 操作非常灵活，支持一个很有趣的特性，就是负值索引，比如一个 n 个元素的 list: lst[n]，那么 lst[-1]就是 lst[n-1]。作为对灵活性的代价，Python 对插入点的确定就不能像 STL 中 vector 那样直截了当，必须处理负数的情形。

可以看到，不管你插入在什么位置，对于 Python 来说，都是合法的，它会自己调整插入的位置。在确定了插入的位置之后，Python 会在代码清单 4-3 的[3]处开始搬动元素，将插入点之后的所有元素向下挪动一个位置，这样，在插入点就能空出一个位置来。一旦搬移元素的工作完成，实际上也就是大功告成了，我们想插入的元素有了容身之地了。

熟悉 C++的读者一定看出来了，这种处理插入的方法实际上与 C++中的 vector 完全一致。没错，正如我们前面提到的，Python 中的 PyListObject 对象与 C++中的 vector 是非常相似的，相反，它和 C++中的 list 却是大相径庭的。

Python 进行元素插入的动作流程如图 4-4 所示：

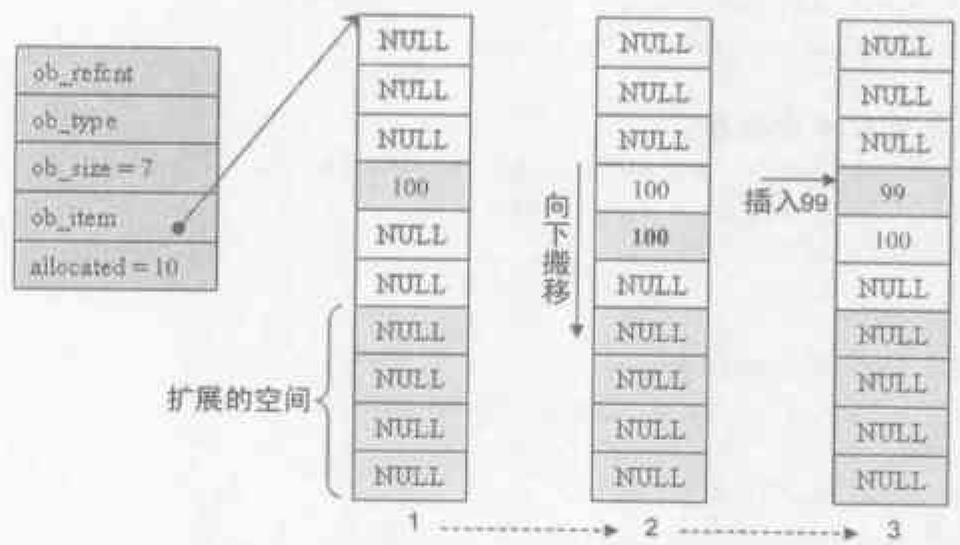



图 4-4 通过 insert 向 PyListObject 对象中插入元素

值得注意的是，通过与 vector 类似的内存管理机制，现在，PyListObject 的 allocated 已经变成 10 了，而 ob_size 却只有 7。

在 Python 中，list 还有另一个被广泛使用的插入操作 append。这个操作与上面所描述的插入操作非常类似：

```
[listobject.c]
//Python 提供的 C API
int PyList_Append(PyObject *op, PyObject *newitem)
{
    if (PyList_Check(op) && (newitem != NULL))
        return app1((PyListObject *)op, newitem);
    return -1;
}

//与 append 对应的 C 函数
static PyObject* listappend(PyListObject *self, PyObject *v)
{
    if (app1(self, v) == 0)
        Py_RETURN_NONE;
    return NULL;
}

static int app1(PyListObject *self, PyObject *v)
{
    int n = PyList_GET_SIZE(self);
    ......
    if (list_resize(self, n+1) == -1)
        return -1;

    Py_INCREF(v);
    PyList_SET_ITEM(self, n, v); //这里是设置操作
    return 0;
}
```

只是需要注意的是，在进行 append 动作的时候，添加的元素是添加在第 ob_size+1 个位置上的（即 list[ob_size] 处），而不是第 allocated 个位置上。图 4-5 展示了 append 元素 101 之后的 PyListObject 对象：

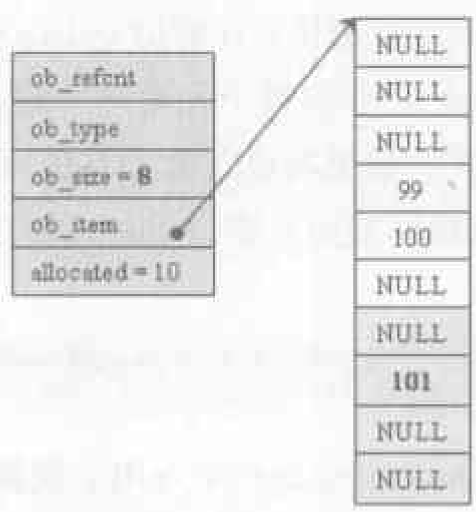



图 4-5 通过 append 向 PyListObject 对象中插入元素

在 appl 中调用 list_resize 时，由于 newsize（8）在 5 和 10 之间，所以不需要再分配内存空间。直接将 101 放置在第 8 个位置上即可。

## 4.2.4 删除元素

对于一个容器而言，除了创建、设置、插入这些操作，至少还应该有删除操作才算完整。倘若不然，只有插入元素的话，再大的容器迟早有一天会被撑爆的，PyListObject 自然也不会例外。图 4-6 展示了一个使用 PyListObject 中删除元素功能的例子。

```
>>> lst = [1, 2, 3, 4]
>>> lst
[1, 2, 3, 4]
>>> lst.remove(3)
>>> lst
[1, 2, 4]
```

图 4-6 删除元素的例子

当 Python 执行 lst.remove(3)时，PyListObject 中的 listremove 操作会被激活：

```
[listobject.c]
static PyObject * listremove(PyListObject *self, PyObject *v)
{
    int i;
    for (i = 0; i < self->ob_size; i++) {
        //比较 list 中的元素与待删除的元素 v
        int cmp = PyObject_RichCompareBool(self->ob_item[i], v, Py_EQ);
        if (cmp > 0) {
            if (list_ass_slice(self, i, i+1,(PyObject *)NULL) == 0)
                Py_RETURN_NONE;
            return NULL;
        }
        else if (cmp < 0)
            return NULL;
    }
    PyErr_SetString(PyExc_ValueError, "list.remove(x): x not in list");
    return NULL;
}
```

Python 会对整个列表进行遍历，在遍历 PyListObject 中所有元素的过程中，将待删除的元素与 PyListObject 中的每个元素一一进行比较，比较操作是通过 PyObject_RichCompareBool 完成的，如果其返回值大于 0，则表示列表中的某个元素与待删除的元素匹配。一旦在列表中发现匹配的元素，Python 将立即调用 list_ass_slice 删除该元素。其函数原形如下：

```
int list_ass_slice(PyListObject *a, Py_ssize_t ilow, Py_ssize_t ihigh,
  PyObject *v)
```

list_ass_slice 实际上并非是一个专用于删除操作函数，它的完整功能如下：

- `a[ilow:ihigh] = v if v != NULL.`
- `del a[ilow:ihigh] if v == NULL.`

可见，这个家伙实际上有着 replace 和 remove 两种语义，决定使用哪种语义的，正是最后一个参数 v。图 4-7 展示了当 Python 内部调用这个函数时，会发生的动作。

```
>>> l = [1, 2, 3, 4]
>>> l[1:3] = ['a', 'b']
>>> l
[1, 'a', 'b', 4]
>>>
>>> l[1:2] = []
>>> l
[1, 'b', 4]
>>>
```

图 4-7 list_ass_slice 的不同语义

当执行 `l[1:3] = ['a', 'b']`时，Python 内部就调用了 `list_ass_slice`，而其参数为 `ilow=1, ihigh=3, v=['a', 'b']`。

而当 `list_ass_slice` 的参数 v 为 NULL 时，Python 就会将默认的 replace 语义替换为 remove 语义，删除[ilow, ihigh]范围内的元素，这正是 listremove 期望的动作。同样，在图 4-7 中，我们通过 `l[1:2] = []`的执行看到了这一点，对于这个表达式语句，Python 内部调用的正是 `list_ass_slice(l, 1, 2, NULL)`。

对于 `list_ass_slice` 的具体实现，这里就不过多地深入了，有兴趣的读者可以参考 Python 的源代码。在 `list_ass_slice` 中，当进行元素删除动作时，实际上是通过 memmove 简单地搬移内存来实现的。这就意味着，当调用 list 的 remove 操作删除 list 中的元素时，一定会触发内存搬移的动作，这一点跟 C++中的 vector 是完全一致的，而与 C++中的 list 则完全不同。

图 4-8 展示了删除元素 100 的过程。

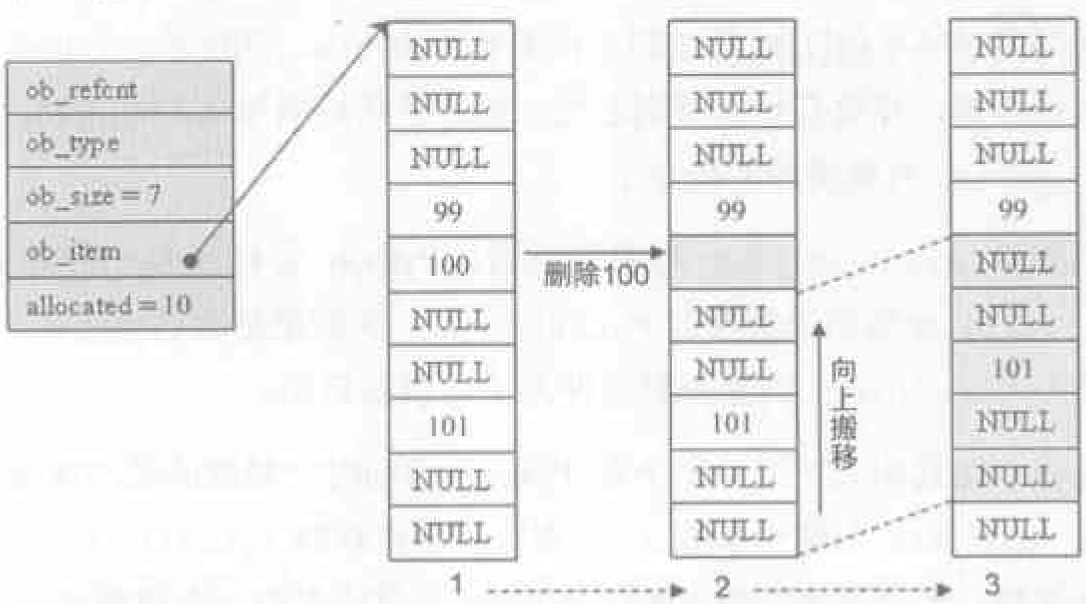



图 4-8 在 PyListObject 中删除 100

# 4.3 PyListObject 对象缓冲池

还记得吗，刚才我们按下了一个有趣的话题。没错，就是那个缓冲池，free_lists。现在，是揭开它神秘面纱的时候了。我们想知道的问题是：free_lists 中所缓冲的 PyListObject 对象是从哪里获得的，是在何时创建的？答案就是在一个 PyListObject 被销毁的过程中（见代码清单 4-4）。

**代码清单 4-4**

```
[listobject.c]
static void list_dealloc(PyListObject *op)
{
    int i;
    //[1] : 销毁 PyListObject 对象维护的元素列表
    if (op->ob_item != NULL) {
        i = op->ob_size;
        while (--i >= 0) {
            Py_XDECREF(op->ob_item[i]);
        }
        PyMem_FREE(op->ob_item);
    }
    //[2] : 释放 PyListObject 自身
    if (num_free_lists < MAXFREELISTS && PyList_CheckExact(op))
        free_lists[num_free_lists++] = op;
    else
        op->ob_type->tp_free((PyObject *)op);
}
```

在创建一个新的 list 时，我们看到创建过程实际分离为两步，首先创建 PyListObject 对象，然后创建 PyListObject 对象所维护的元素列表。与之对应，在销毁一个 list 时，销毁的过程也是分离的，首先 Python 会在代码清单 4-4 的[1]处销毁 PyListObject 对象所维护的元素列表，然后在代码清单 4-4 的[2]处释放 PyListObject 对象自身。

代码清单 4-4 的[1]处所做的工作无非是为 list 中的每一个元素改变其引用计数，然后将内存释放，并没有什么特别之处。而到了代码清单 4-4 的[2]处，有趣的东西出现了，PyListObject 对象缓冲池现身了。

在删除 PyListObject 对象自身时，Python 会检查我们开始提到的那个缓冲池，free_lists，查看其中缓存的 PyListObject 的数量是否已经满了。如果没有，就将该待删除的 PyListObject 对象放到缓冲池中，以备后用。

现在一切真相大白了，那个在 Python 启动时空荡荡的缓冲池原来都是被本应该死去的 PyListObject 对象给填充了，在以后创建新的 PyListObject 的时候，Python 会首先唤醒这些已经“死去”的 PyListObject，又给它们一个重新做“人”的机会☺。但是，需要指出，这里缓冲的仅仅是 PyListObject 对象，而没有这个对象曾经拥有的 PyObject*

元素列表，因为这些 PyObject*指针的引用计数已经减少了，这些指针所指的对象都要各奔前程，或生存，或毁灭，不再被 PyListObject 所给予的那个引用计数所束缚。PyListObject 如果继续维护一个指向这些对象指针的列表，就可能产生悬空指针的问题。所以，PyObject*列表所占用的空间必须归还给系统。

当然，我们实际上可以将 PyListObject 对象所维护的元素列表保留，即在代码清单 4-4 的[1]处仅仅调整引用计数，并将列表中的各个元素都设置为 NULL，却并不释放元素列表的内存空间。但是如此一来，这些已经被释放的内存并不会归还给系统堆，这就意味着除了 PyListObject 对象自身，没有人能再使用这些内存。虽然保留元素列表可以在创建 list 时免去创建元素列表的开销，但是 Python 为了避免过多消耗系统内存，采取了将元素列表的内存归还给系统堆的做法，以时间换取空间。

图 4-9 显示了如果删除我们在前面创建的那个 list，PyListObject 对象缓冲池的示意图。

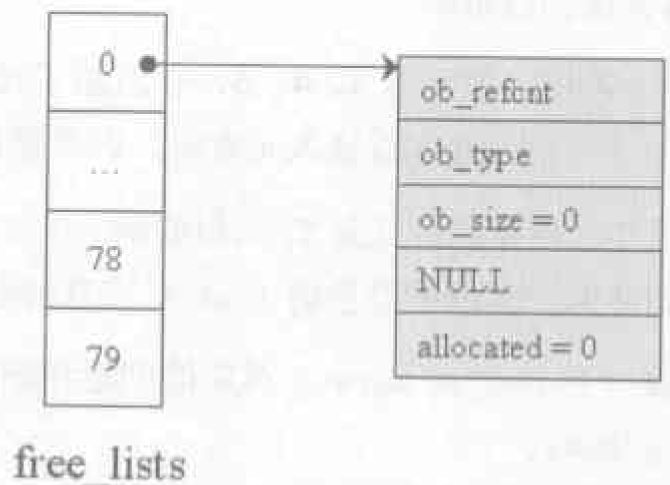



图 4-9 PyListObject 对象缓冲池

在 Python 下一次创建新的 list 时，这个 PyListObject 对象将重新被唤醒，重新分配 PyObject*元素列表占用的内存，重新拥抱新的对象。放眼四周，曾经所拥有过的那些对象，有的已经容颜苍老，有的已经烟消云散，身为它们曾经的头儿，是否有一种“物是人非事事休，欲语泪先流”的感慨呢？☺

## 4.4 Hack PyListObject

首先我们来观察在 PyListObject 中维护的元素数量变化时，PyListObject 中 ob_size 和 allocated 两个变量的变化情况，从中窥见 PyListObject 对内存的使用和管理。

在 PyListObject 的输出操作 list_print 中，我们添加了如下代码，以观察 PyList-

Object 对内存的管理：

```
printf("\nallocated=%d, ob_size=%d\n", op->allocated, op->ob_size);
```

观察结果如图 4-10 所示。

| | | | |
|---|---|---|---|
| >>> lst = [1]<br>>>> lst<br>[1]<br>allocated=1, ob_size=1 | >>> lst.append(2)<br>>>> lst<br>[1, 2]<br>allocated=5, ob_size=2 | >>> lst.append(3)<br>>>> lst<br>[1, 2, 3]<br>allocated=5, ob_size=3 | >>> lst.remove(2)<br>>>> lst<br>[1, 3]<br>allocated=5, ob_size=2 |

图 4-10 观察 PyListObject 中的内存管理

首先创建一个包含一个元素的 list，这时 ob_size 和 allocated 都是 1。这时 list 中拥有的所有内存空间都已被使用完，所以下次插入元素时就一定会调整 list 的内存空间了。

随后向 list 末尾追加元素 2，可以看到，调整内存空间的动作发生了。allocated 变成了 5，而 ob_size 则变成了 2，这里明确地显示出了 PyListObject 所采用的与 C++ 中 vector 一样的内存缓冲池策略。

如果继续向 list 末尾追加元素 3、4、5，在追加了元素 5 之后，list 所拥有的内存空间又被使用完了，下一次再追加或插入元素时，内存空间调整的动作又会再一次发生。

如果在追加元素 3 之后就删除元素 2，可以看到，ob_size 发生了变化，而 allocated 则不发生变化，它始终如一地维护着当前 list 所拥有的全部内存数量。

接下来我们观察一下 PyListObject 对象的创建和删除对于 Python 维护的 PyListObject 对象缓冲池的影响。

```
list1 = [1]
print list1
list2 = [1]
print list1
list3 = [1]
print list1
del list2
print list1
del list3
print list1
```

```
[1]
num_free_lists=3

[1]
num_free_lists=2

[1]
num_free_lists=1

[1]
num_free_lists=2

[1]
num_free_lists=3
```

图 4-11 观察 PyListObject 对象缓冲池的使用

这次为了消除 Python 交互环境执行时对 PyListObject 对象缓冲池的影响，我们通过执行 py 脚本文件来观察。从图 4-11 中可以看到，当创建新的 PyListObject 对象时，如果缓冲池中有可用的 PyListObject 对象，则会使用缓冲池中的对象；而在销毁一个 PyListObject 对象时，确实将这个对象放到了缓冲池中。

CHAPTER

# 第 5 章

# Python 中的 Dict 对象

元素和元素之间通常可能存在某种联系，这种神秘的联系使本来毫不相关的两个元素被捆绑在一起，而别的元素则被排斥在外。比如对于“2 倍”这样一种联系，6 和 3 就是这样的两个元素，而 4 和 2 同样也是被这种联系关联起来的一对元素。

为了刻画某种对应关系，现代的编程语言通常都在语言级或标准库中提供某种关联式的容器。关联式的容器中存储着一对对符合该容器所代表的关联规则的元素对。关联式容器中的元素对通常是以（键（key）或值（value））这样的形式存在。比如在一个表示“2 倍”关系的关联容器中，(3，6)，(2，4) 就是容器中的两个元素对。其中 3 就是一个“键”，当寻找到 3 之后，我们就能很轻松地获得与 3 有着“2 倍”联系的另一元素。

关联容器的设计总会极大地关注键的搜索效率，因为通常我们使用关联容器，都是希望根据我们手中已有的某个元素来快速获得与之有某种联系的另一元素。一般而言，关联容器的实现都会基于设计良好的数据结构。比如 C++的 STL 中的 `map` 就是一种关联容器，`map` 的实现基于 RB-tree（红黑树）。RB-tree 是一种平衡二元树，能够提供良好的搜索效率，理论上，其搜索的时间复杂度为 O（$\log_2 N$）。

Python 中同样提供关联式容器，即 `PyDictObject` 对象（也称 `dict`）。与 `map` 不同的是，`PyDictObject` 对搜索的效率要求极其苛刻，这也是因为 `PyDictObject` 对象在 Python 本身的实现中被大量地采用。比如 Python 会通过 `PyDictObject` 来建立执行 Python 字节码的运行环境，其中会存放变量名和变量值的元素对，通过查找变量名获得变量值。因此，`PyDictObject` 没有如 `map` 一样采用平衡二元树，而是采用了散列表（hash table），因为理论上，在最优情况下，散列表能提供 O（1）复杂度的搜索效率。

# 5.1 散列表概述

散列表的基本思想，是通过一定的函数将需搜索的键值映射为一个整数，将这个整数视为索引值去访问某片连续的内存区域。看一个简单的例子，如图 5-1 所示，有 10 个整数 1，2，…，10，其依次对应 a，b，…，j。申请一块连续内存，并依次存储 a，b，…，j：

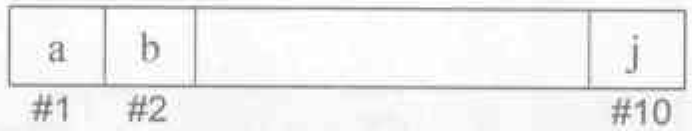



图 5-1 散列表的例子

当需要寻找与 2 对应的字母时，只需通过一定的函数将其映射为整数，显然，2 本身就是一个整数，我们可以使用这样的映射函数 `f (n) = n`，那么 2 的映射值就是 2 本身。然后访问这片连续内存的第 2 个位置，就能得到与 2 对应的字母 b。

对散列表这种数据结构的采用是以加速键的搜索过程为终极目标的，于是，将元素映射为整数的过程对于 Python 中 `dict` 的实现就显得尤为关键。用于映射的函数称为散列函数（hash function），而映射后的值称为元素的散列值（hash value）。在散列表的实现中，所选择的散列函数的优劣将直接决定所实现的散列表的搜索效率的高低。

在使用散列表的过程中，不同的对象经过散列函数的作用，可能被映射为相同的散列值。而且随着需要存储的数据的增多，这样的冲突就会发生得越来越频繁。散列冲突是散列技术与生俱来的问题。这里需要提到一个散列表中与散列冲突相关的概念——装载率。装载率是散列表中已使用空间和总空间的比值。举例来说，如果散列表一共可以容纳 10 个元素，而当前已经装入了 6 个元素，那么装载率就是 6/10。研究表明，当散列表的装载率大于 2/3 时，散列冲突发生的概率就会大大增加。

有很多方法可以用来解决产生的散列冲突问题，比如开链法，这是 SGI STL 中的 hash table 所采用的方法，而 Python 中所采用的是另一种方法，即开放定址法。

当产生散列冲突时，Python 会通过一个二次探测函数 `f`，计算下一个候选位置 `addr`，如果位置 `addr` 可用，则可将待插入元素放到位置 `addr`；如果位置 `addr` 不可用，则 Python 会再次使用探测函数 `f`，获得下一个候选位置，如此不断探测，总会找到一个可用的位置。

这样，通过多次使用探测函数 `f`，从一个位置出发就可依次到达多个位置，我们认为这些位置形成了一个 **“冲突探测链”**（或简称**探测序列**）。当需要删除某条探测链上的某个元素时，问题就产生了。假如这条链的首元素位置为 a，尾元素的位置为 c，现在需要删除中间的某个位置 b 上的元素。如果直接将位置 b 上的元素删除，则会导致探测链的断裂，造成严重的后果。想象一下，在下一次搜索位置 c 的元素时，会从位置 a 开始，通过探测

函数，沿着探测链，一步一步向位置 c 靠近，但是在到达位置 b 时，发现这个位置上的元素不属于这个探测链，因此探测函数会以为探测链在这里结束，导致不能到达位置 c，自然也就不能搜索到位置 c 上的元素，所以结果是搜索失败。而实际上，待搜索的元素确实存在于散列表中。

所以，在采用开放定址的冲突解决策略的散列表中，删除某条探测链上的元素时不能进行真正的删除，而是进行一种 **“伪删除”** 操作，必须要让该元素还存在于探测链上，担当承前启后的重任。对于这种伪删除技术，我们在分析 Python 中的 `PyDictObject` 对象时会详细讨论。

# 5.2 PyDictObject

## 5.2.1 关联容器的 entry

在本章此后的描述中，我们将把关联容器中的一个（键，值）元素对称为一个 entry 或 slot。在 Python 中，一个 entry 的定义如下：

```
[dictobject.h]
typedef struct {
    Py_ssize_t me_hash;      /* cached hash code of me_key */
    PyObject *me_key;
    PyObject *me_value;
} PyDictEntry;
```

可以看到，在 `PyDictObject` 中其实存放的都是 `PyObject*`，这也是 Python 中的 `dict` 什么都能装得下的原因，因为在 Python 中，无论什么东西归根结底都是一个 `PyObject` 对象。

在 `PyDictEntry` 中，`me_hash` 域存储的是 `me_key` 的散列值，利用一个域来记录这个散列值可以避免每次查询的时候都要重新计算一遍散列值。

在 Python 中，在一个 `PyDictObject` 对象生存变化的过程中，其中的 `entry` 会在不同的状态间转换。`PyDictObject` 中 `entry` 可以在 3 种状态间转换：Unused 态、Active 态和 Dummy 态。

- 当一个 `entry` 的 `me_key` 和 `me_value` 都是 `NULL` 时，`entry` 处于 Unused 态。Unused 态表明目前该 `entry` 中并没有存储（key，value）对，而且在此之前，也没有存储过它们。每一个 `entry` 在初始化的时候都会处于这种状态，而且只有在 Unused 态下，`entry` 的 `me_key` 域才会为 `NULL`。
- 当 `entry` 中存储了一个（key，value）对时，`entry` 便转换到了 Active 态。在 Active 态下，`me_key` 和 `me_value` 都不能为 `NULL`。更进一步地说，`me_key` 不能是 `dummy` 对象。

- 当 entry 中存储的（key，value）对被删除后，entry 的状态不能直接从 Active 态转为 Unused 态，否则会如我们前面提到的，导致冲突探测链的中断。相反，entry 中的 me_key 将指向 dummy 对象（这个 dummy 对象究竟为何方神圣，后面我们会详细考察），entry 进入 Dummy 态，这就是我们前面提到的"伪删除"技术。当 Python 沿着某条冲突链搜索时，如果发现一个 entry 处于 Dummy 态，说明目前该 entry 虽然是无效的，但是其后的 entry 可能是有效的，是应该被搜索的。这样，就保证了冲突探测链的连续性。

图 5-2 展示了 entry 的 3 种可能状态，以及它们之间的转换关系：

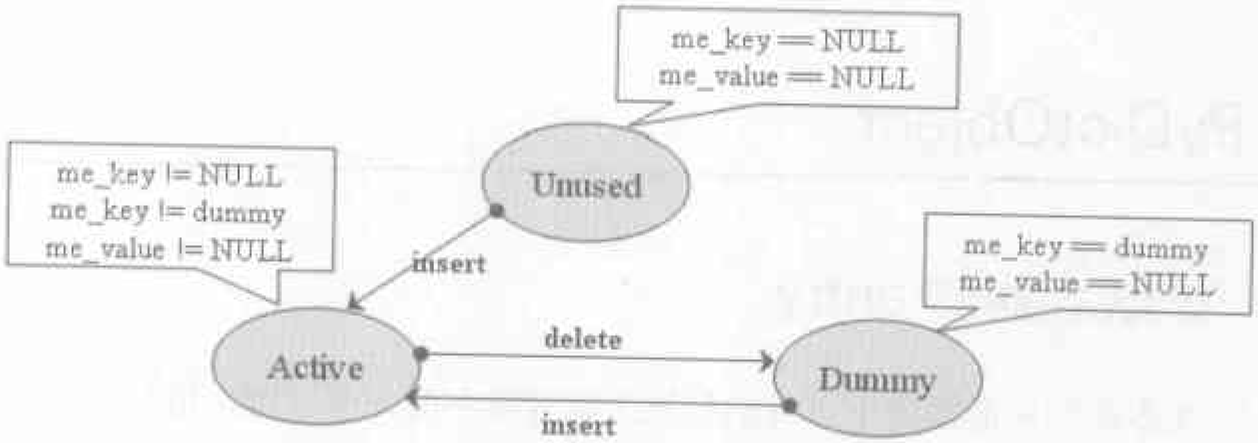



图 5-2 PyDictObject 中 entry 的状态转换图

## 5.2.2 关联容器的实现

在 Python 中，关联容器是通过 PyDictObject 对象来实现的。而一个 PyDictObject 对象实际上是一大堆 entry 的集合，总控这些集合的结构如下：

```
[dictobject.h]
#define PyDict_MINSIZE 8
typedef struct _dictobject PyDictObject;
struct _dictobject {
    PyObject_HEAD
    Py_ssize_t ma_fill;  //元素个数: Active + Dummy
    Py_ssize_t ma_used;  //元素个数: Active
    Py_ssize_t ma_mask;
    PyDictEntry *ma_table;
    PyDictEntry *(*ma_lookup)(PyDictObject *mp, PyObject *key, long hash);
    PyDictEntry ma_smalltable[PyDict_MINSIZE];
};
```

从注释中可以清楚地看到，ma_fill 域中维护着从 PyDictObject 对象创建开始直到现在，曾经及正处于 Active 态的 entry 个数，而 ma_used 则维护着当前正处于 Active 态的 entry 的数量。

在 PyDictObject 定义的最后，有一个名为 ma_smalltable 的 PyDictEntry 数组。这个数组意味着当创建一个 PyDictObjec 对象时，至少有 PyDict_MINSIZE 个 entry 被同时创建。在 dictobject.h 中，这个值被设定为 8，这个值被认为是通过大量的实验得出的最佳值。它既不会太浪费内存空间，又能很好地满足 Python 内部大量使用 PyDictObject 的环境的需求，不需要在使用的过程中再次调用 malloc 申请内存空间。当然，我们可以自己改变这个值来调节 Python 的运行效率。

PyDictObject 中的 ma_table 域是关联对象的关键所在，这个类型为 PyDictEntry* 的变量将指向一片作为 PyDictEntry 集合的内存的开始位置。当一个 PyDictObject 对象是一个比较小的 dict 时，即 entry 数量少于 8 个，ma_table 域将指向 ma_smalltable 这与生俱来的 8 个 entry 的起始地址。而当 PyDictObject 中的 entry 数量超过 8 个时，Python 认为这家伙是一个大 dict 了，将会申请额外的内存空间，并将 ma_table 指向这块空间。这样，无论何时，ma_table 域都不会为 NULL，这带来了一个好处，不用在运行时一次又一次地检查 ma_table 的有效性，因为 ma_table 总是有效的。

图 5-3 分别显示了 Python 中的"大"，"小"两种 dict：

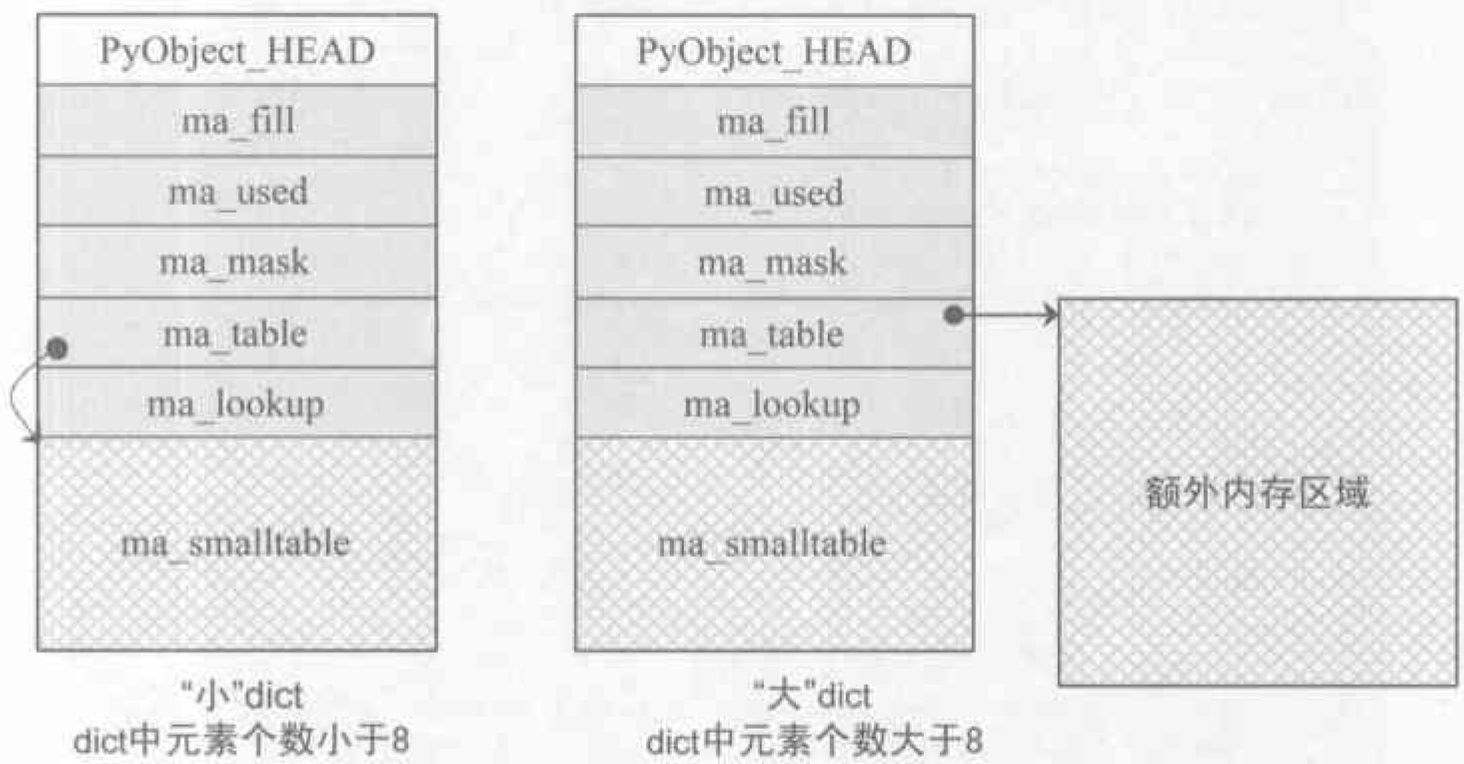



图 5-3 PyDictObject 中 ma_table 的两种可能状态

最后，PyDictObject 中的 ma_mask 实际上记录了一个 PyDictObject 对象中所拥有的 entry 的数量。至于这家伙为什么不叫 ma_size 这么好听的名字，偏偏要特立独行，取一个 ma_mask 这样莫名其妙的名字，此乃后话，这里先按下不表。同样被按下不表的还有 ma_lookup☺。

# 5.3 PyDictObject 的创建和维护

## 5.3.1 PyDictObject 对象创建

Python 内部通过 PyDict_New 来创建一个新的 dict 对象（见代码清单 5-1）。

代码清单 5-1

```
[dictobject.c]
typedef PyDictEntry dictentry;
typedef PyDictObject dictobject;

#define INIT_NONZERO_DICT_SLOTS(mp) do {                    \
    (mp)->ma_table = (mp)->ma_smalltable;                  \
    (mp)->ma_mask = PyDict_MINSIZE - 1;                    \
    } while(0)

#define EMPTY_TO_MINSIZE(mp) do {                           \
    memset((mp)->ma_smalltable, 0, sizeof((mp)->ma_smalltable));    \
    (mp)->ma_used = (mp)->ma_fill = 0;                     \
    INIT_NONZERO_DICT_SLOTS(mp);                           \
    } while(0)

PyObject* PyDict_New(void)
{
    register dictobject *mp;
    //[1]: 自动创建 dummy 对象
    if (dummy == NULL) {
        dummy = PyString_FromString("<dummy key>");
    }

    if (num_free_dicts)
    {
        …… //[2]: 使用缓冲池
    }
    else
    {
        //[3]: 创建 PyDictObject 对象
        mp = PyObject_GC_New(dictobject, &PyDict_Type);
        EMPTY_TO_MINSIZE(mp);
    }
    mp->ma_lookup = lookdict_string;
    return (PyObject *)mp;
}
```

值得注意的是，第一次调用 PyDict_New 时，在代码清单 5-1 的[1]处会创建前面提到的那个 dummy 对象。原来 dummy 竟然是一个 PyStringObject 对象，实际上，它仅仅是用来作为一种指示标志，表明该 entry 曾被使用过，且探测序列下一个位置的 entry 有可能是有效的，从而防止探测序列中断。

从 num_free_dicts 可以看出，Python 中 dict 的实现同样使用了缓冲池。我们把代

码清单 5-1 的[2]处对缓冲池的使用的讨论放到后面。

如果 PyDictObject 对象的缓冲池不可用，那么 Python 将首先从系统堆中为新的 PyDictObject 对象申请合适的内存空间，然后会通过两个宏完成对新生的 PyDictObject 对象的初始化工作：

- EMPTY_TO_MINSIZE：将 ma_smalltable 清零，同时设置 ma_size 和 ma_fill，当然，在一个 PyDictObject 对象刚被创建的时候，这两个变量都应该是 0。
- INIT_NONZERO_DICT_SLOT：将 ma_table 指向 ma_smalltable，并设置 ma_mask 为 7。

可以看到，ma_mask 的初始值为 PyDict_MINSIZE-1，确实与一个 PyDictObject 对象中 entry 的数量有关。在创建过程的最后，将 lookdict_string 赋给了 ma_lookup。正是这个 ma_lookup 指定了 PyDictObject 在 entry 集合中搜索某一特定 entry 时需要进行的动作，在 ma_lookup 中，包含了散列函数和发生冲突时二次探测函数的具体实现，众所周知，它是 PyDictObject 的搜索策略。

## 5.3.2 PyDictObject 中的元素搜索

Python 为 PyDictObject 对象提供了两种搜索策略，lookdict 和 lookdict_string。实际上，这两种策略使用的是相同的算法，lookdict_string 只是 lookdict 的一种针对 PyStringObject 对象的特殊形式。以 PyStringObject 对象作为 PyDictObject 对象中 entry 的键在 Python 中是如此的广泛和重要，所以 lookdict_string 也就成为了 PyDictObject 创建时所默认采用的搜索策略。

既然 lookdict_string 是 lookdict 针对 PyStringObject 对象的特殊形式，那么我们首先就来剖析一下 dict 中的通用搜索策略 lookdict，而不是考虑 Python 的默认搜索策略。一旦清晰地了解了通用搜索策略，lookdict_string 也就一目了然了（见代码清单 5-2）。

**代码清单 5-2**

```
[dictobject.c]
static dictentry* lookdict(dictobject *mp, PyObject *key, register long
  hash)
{
    register size_t i;
    register size_t perturb;
    register dictentry *freeslot;
    register size_t mask = mp->ma_mask;
    dictentry *ep0 = mp->ma_table;
    register dictentry *ep;

    register int restore_error;
```

```
    register int checked_error;
    register int cmp;
    PyObject *err_type, *err_value, *err_tb;
    PyObject *startkey;

    //[1]: 散列，定位冲突探测链的第一个 entry
    i = hash & mask;
    ep = &ep0[i];

    //[2]:
    //1. entry 处于 Unused 态
    //2. entry 中的 key 与待搜索的 key 匹配
    if (ep->me_key == NULL || ep->me_key == key)
        return ep;

    //[3]: 第一个 entry 处于 Dummy 态，设置 freeslot
    if (ep->me_key == dummy)
        freeslot = ep;
    else
    {
        //[4]: 检查 Active 态 entry
        if (ep->me_hash == hash)
        {
            startkey = ep->me_key;
            cmp = PyObject_RichCompareBool(startkey, key, Py_EQ);
            if (cmp > 0)
                return ep;
        }
        freeslot = NULL;
    }
    ……
}
```

这里列出的代码对 Python 的源码进行了一些改动，目的是为了简明地展示搜索的过程，同时需要指出，这里列出的只是 Python 对冲突链上第一个 entry 所进行的动作。

PyDictObject 中维护的 entry 的数量是有限的，比如 100 个或 1000 个。而传入 lookdict 中的 key 的 hash 值却不一定会在这个范围内，所以这就要求 lookdict 将 hash 值映射到某个 entry 上去。lookdict 采取的策略非常简单，直接将 hash 值与 entry 的数量做一个与操作，结果自然落在 entry 的数量之下。代码清单 5-2 的[1]处实现了这个过程，由于 ma_mask 会被用来进行大量的与操作，所以这个与 entry 数量相关的变量被命名为 ma_mask，而不是 ma_size。

这里需要说明一下，无论是 lookdict_string 还是 lookdict，永远都不会返回 NULL，如果在 PyDictObject 中搜索不到待查找的 key，同样会返回一个 entry，这个 entry 的 me_value 为 NULL。这个 entry 指示搜索失败，并且该 entry 是一个空闲的 entry，马上就可被 Python 所使用。

在搜索的过程中，代码清单 5-2 的[3]处所操纵的 freeslot 是一个重要的变量。如果

在探测链中的某个位置上，entry 处于 Dummy 态，那么如果在这个序列中搜索不成功，就会返回这个处于 Dummy 态的 entry。我们知道，处于 Dummy 态的 entry 其 me_value 是为 NULL 的，所以这个返回结果指示了搜索失败；同时，返回的 entry 也是一个可以立即被使用的 entry，因为 Dummy 态的 entry 并没有维护一个有效的（key，value）对。这个 freeslot 正是用来指向探测序列中第一个处于 Dummy 态的 entry，如果搜索失败，freeslot 就会挺身而出，提供一个能指示失败并立即可用的 entry。当然，如果探测序列中并没有 Dummy 态 entry，搜索失败时，一定是在一个处于 Unused 态的 entry 上结束搜索过程的，这时会返回这个处于 Unused 态的 entry，同样是一个能指示失败且立即可用的 entry。

在 Python 对 dict 的搜索中，更进一步的，在 dict 的实现中，有一个抽象的概念，虽然我们至今没有提及，但是却至关重要。我们说在 dict 中的搜索过程是要在 dict 的所有 entry 中寻找一个这样的 entry，其 key 与待搜索的 key **相同**。那么什么是相同呢？如何定义相同？如果没有确定这个概念，那么搜索就无从谈起，dict 也就无从谈起。

在 Python 的 dict 中，“相同”这个概念实际上包含两层含义：1. 引用相同；2. 值相同。Python 中的 dict 正是建立在这两层含义之上的。所谓引用相同，是指两个符号引用的是内存中的同一个地址，这一点的检查是由代码清单 5-2 的[2]处的“ep->me_key == key”所完成的；而所谓的值相同，是说两个 PyObject*指针实际上指向了不同的对象，即内存中的不同位置，但是这两个对象的值相同。

举一个简单的例子，在前面我们已经看到，在整数对象中，小整数对象是共享的，而大整数对象并不是共享的，当我们多次创建相同的大整数对象时，虽然其值相同，然而 Python 创建的却是不同的对象，考虑如图 5-4 所示的 Python 代码：

```
>>> d = {}
>>> d[9876] = 'Python'
>>> print d[9876]
Python
>>>
```

图 5-4 在 dict 中搜索整数

这里出现了两个整数对象 9876，在第三行调用 print d[9876]时，Python 会首先到 d 中搜索键为 9876 的 entry，显然，在 lookdict 中，代码清单 5-2 的[2]处的引用相同检查是不会成功的，但是如果这就意味着该 entry 不存在，那简直会让人匪夷所思，因为在图 5-4 中我们看到，这个 entry 明明是存在的。这就是“值相同”这条规则存在的意义。

在 lookdict 中，代码清单 5-2 的[4]处完成了两个 key 的值检查。值检查的过程首先会检查两个对象的 hash 值是否相同，如果不相同，则其值也一定不相同，不用再继续下去了；而如果 hash 值相等，那么 Python 将通过 PyObject_RichCompareBool 进行比较，注

意这个函数，它的原型为：

```
int PyObject_RichCompareBool(PyObject *v, PyObject *w, int op)
```

这是 Python 提供的一个相当典型的比较操作，我们可以自己指定比较操作的类型，当(v op w)成立时，返回 1；当(v op w)不成立时，返回 0；如果在比较中发生错误，则返回-1。在代码清单 5-2 的[4]处，lookdict 指定了 Py_EQ，这将指示 PyObject_RichCompareBool 进行相等比较操作。

下面我们来总结一下 lookdict 中进行第一次检查时所进行的主要动作，如代码清单 5-2 中的[1]、[2]、[3]、[4]所示。

[1] 根据 hash 值获得 entry 的索引，这是冲突探测链上第一个 entry 的索引。

[2] 在两种情况下，搜索结束：

- entry 处于 Unused 态，表明冲突探测链搜索完成，搜索失败；
- ep->me_key == key，表明 entry 的 key 与待搜索的 key 匹配，搜索成功。

[3] 若当前 entry 处于 Dummy 态，设置 freeslot。

[4] 检查 Active 态 entry 中的 key 与待查找的 key 是否"值相同"，若成立，搜索成功。

在以上的剖析中，我们考察的是根据 hash 值获得的冲突探测链上第一个 entry 与待查找的元素的比较。实际上，由于 entry 对应于某一个散列值，几乎都有一个冲突探测链与之对应，所以我们现在只是考察了所有候选 entry 中的第一个 entry，万里长征仅仅迈出了第一步。

如果冲突探测链上第一个 entry 的 key 与待查找的 key 不匹配，那么很自然地，lookdict 会沿着探测链，顺藤摸瓜，依次比较探测链上的 entry 与待查找的 key（见代码清单 5-3）。

**代码清单 5-3**

```
[dictobject.c]
static dictentry* lookdict(dictobject *mp, PyObject *key, register long
hash)
{
    register int i;
    register unsigned int perturb;
    register dictentry *freeslot;
    register unsigned int mask = mp->ma_mask;
    dictentry *ep0 = mp->ma_table;
    register dictentry *ep;
    register int cmp;
    PyObject *startkey;
    ......
    for (perturb = hash; ; perturb >>= PERTURB_SHIFT)
    {
        //[5]: 寻找探测链上下一个 entry
        i = (i << 2) + i + perturb + 1;
        ep = &ep0[i & mask];
```

```
        //[6]: 到达 Unused 态 entry，搜索失败
        if (ep->me_key == NULL)
            return freeslot == NULL ? ep : freeslot;
        //[7]: 检查“引用相同”是否成立
        if (ep->me_key == key)
            return ep;
        //[8]: 检查“值相同”是否成立
        if (ep->me_hash == hash && ep->me_key != dummy)
        {
            startkey = ep->me_key;
            cmp = PyObject_RichCompareBool(startkey, key, Py_EQ);
            if (cmp > 0)
                return ep;
        }
        //[9]: 设置 freeslot
        else if (ep->me_key == dummy && freeslot == NULL)
            freeslot = ep;
    }
}
```

我们已经清楚地了解了 lookdict 检查冲突探测链上的第一个 entry 时所进行的动作，其实对探测链上的其他 entry 也将进行同样的动作，对第一个 entry 和其他 entry 的检查本质上是一样的，我们看一看在遍历探测链时发生 lookdict 所进行的操作，如代码清单 5-3 中的[5]、[6]、[7]、[8]、[9]所示。

[5] 根据 Python 所采用的探测函数，获得探测链中的下一个待检查的 entry。

[6] 检查到一个 Unused 态 entry，表明搜索失败，这时有两种结果：

- 如果 freeslot 不为空，则返回 freeslot 所指 entry；
- 如果 freeslot 为空，则返回该 Unused 态 entry。

[7] 检查 entry 中的 key 与待查找的 key 是否符合“引用相同”规则。

[8] 检查 entry 中的 key 与待查找的 key 是否符合“值相同”规则。

[9] 在遍历过程中，如果发现 Dummy 态 entry，且 freeslot 未设置，则设置 freeslot。

需要特别注意的是，如果搜索成功，那么 ep 一定指向一个有效的 entry，直接返回这个 entry 即可；如果搜索失败，那么此时 ep 指向一个 Unused 态的 entry，我们不能直接返回该 entry，因为有可能在遍历的过程中，已经发现了一个 Dummy 态 entry，这个 entry 实际是一个空闲的 entry，可以被 Python 使用，所以在代码清单 5-3 的[6]处，我们会检查当前 freeslot 是否已经被设置，如果被设置，则不会返回 Dummy 态 entry，而是需要返回 freeslot 所指向的 entry。

到这里，我们已经清晰地了解了 PyDictObject 中的搜索策略，现在可以来看一看 Python 在 PyDict_New 中为 PyDictObject 对象提供的默认搜索策略了（见代码清单 5-4）。

**代码清单 5-4**

```
[dictobject.c]
static dictentry* lookdict_string(dictobject *mp, PyObject *key, register
long hash)
{
    register int i;
    register unsigned int perturb;
    register dictentry *freeslot;
    register unsigned int mask = mp->ma_mask;
    dictentry *ep0 = mp->ma_table;
    register dictentry *ep;
    //[0]: 选择搜索策略
    if (!PyString_CheckExact(key)) {
        mp->ma_lookup = lookdict;
        return lookdict(mp, key, hash);
    }
    //搜索第一阶段: 检查冲突链上第一个 entry
    //[1]: 散列, 定位冲突探测链的第一个 entry
    i = hash & mask;
    ep = &ep0[i];

    //[2]:
    //1. entry 处于 Unused 态
    //2. entry 中的 key 与待搜索的 key 匹配
    if (ep->me_key == NULL || ep->me_key == key)
        return ep;

    //[3]: 第一个 entry 处于 Dummy 态, 设置 freeslot
    if (ep->me_key == dummy)
        freeslot = ep;
    else
    {
        //[4]: 检查 Active 态 entry
        if (ep->me_hash == hash && _PyString_Eq(ep->me_key, key))
        {
            return ep;
        }
        freeslot = NULL;
    }

    //搜索第二阶段: 遍历冲突链, 检查每一个 entry
    for (perturb = hash; ; perturb >>= PERTURB_SHIFT)
    {
        i = (i << 2) + i + perturb + 1;
        ep = &ep0[i & mask];
        if (ep->me_key == NULL)
            return freeslot == NULL ? ep : freeslot;
        if (ep->me_key == key
            || (ep->me_hash == hash && ep->me_key != dummy &&
                _PyString_Eq(ep->me_key, key)))
            return ep;
        if (ep->me_key == dummy && freeslot == NULL)
            freeslot = ep;
    }
}
```

正如我们前面所说，lookdict_string 是一种有条件限制的搜索策略。lookdict_string 背后有一个假设，即待搜索的 key 是一个 PyStringObject 对象。只有在这种假设成立的情况下，lookdict_string 才会被使用。需要特别注意的是，这里只对需要搜索的 key 进行了假设，没有对参与搜索的 dict 做出任何的假设。这就意味着，即使参与搜索的 dict 中所有 entry 的 key 都是 PyIntObject 对象，只要待搜索的 key 是 PyStringObject 对象，都会采用 lookdict_string 进行搜索，_PyString_Eq 将保证能正确处理非 PyStringObject*参数。

在代码清单 5-4 的[0]处，lookdict_string 首先会对这种假设进行确定，检查需要搜索的 key 是否严格对应一个 PyStringObject 对象，只有在检查通过后，才会进行下面的动作；如果检查不通过，那么就会转向 PyDictObject 中的通用搜索策略 lookdict。

可以很清晰地看到，lookdict_string 实际上就是一个 lookdict 对于 PyStringDict 对象的优化版本。我们在本章展示的 lookdict 代码经过了删节，实际上，在 lookdict 中有许多捕捉错误并处理错误的代码，因为 lookdict 面对的是 PyObject*，所以会出现很多意外情况。而在 lookdict_string 中，完全没有了这些处理错误的代码。而另一方面，在 lookdict 中，使用的是非常通用的 PyObject_RichCompareBool，而 lookdict_string 使用的是 _PyString_Eq，要简单很多，这些因素使得 lookdict_string 的搜索效率要比 lookdict 高很多。

那么为什么仅仅是 PyStringObject 对象呢？Python 为什么需要 PyStringObject 对象的一个优化版本，而对 PyIntObject 及其他对象则不闻不问呢？很显然，对于 PyIntObject 对象，我们同样能够写出一个优化的 lookdict_int。

原因在于，Python 自身大量使用了 PyDictObject 对象，用来维护一个名字空间中变量名和变量值之间的对应关系，或是用来在为函数传递参数时维护参数名与参数值的对应关系。这些对象几乎都是用 PyStringObject 对象作为 entry 中的 key，所以 lookdict_string 的意义就显得非常重要了，它对 Python 整体的运行效率都有着重要的影响。

### 5.3.3 插入与删除

PyDictObject 对象中元素的插入动作建立在搜索的基础之上，理解了 PyDictObject 对象中的搜索策略，对于插入动作也就很容易理解了（见代码清单 5-5）。

**代码清单 5-5**

```
[dictobject.c]
static void
insertdict(register dictobject *mp, PyObject *key, long hash, PyObject
*value)
{
```

```
    PyObject *old_value;
    register dictentry *ep;

    ep = mp->ma_lookup(mp, key, hash);
    //[1]: 搜索成功
    if (ep->me_value != NULL) {
        old_value = ep->me_value;
        ep->me_value = value;
        Py_DECREF(old_value);
        Py_DECREF(key);
    }
    //[2]: 搜索失败
    else {
        if (ep->me_key == NULL)
            mp->ma_fill++;
        else
            Py_DECREF(ep->me_key);
        ep->me_key = key;
        ep->me_hash = hash;
        ep->me_value = value;
        mp->ma_used++;
    }
}
```

前面我们提到，搜索操作在成功时，返回相应的处于 Active 态的 `entry`，而在搜索失败时会返回两种不同的结果：一是处于 Unused 态的 `entry`；二是处于 Dummy 态的 `entry`。那么插入操作对应不同的 `entry`，所需要进行的动作显然也是不一样的。对于 Active 的 `entry`，只需要简单地替换 `me_value` 值就可以了；而对于 Unused 或 Dummy 的 `entry`，则需要完整地设置 `me_key`，`me_hash` 和 `me_value`。

在 insertdict 中，正是根据搜索的结果采取了不同的动作，如代码清单 5-5 中的[1]、[2] 所示。

[1] 搜索成功，返回处于 Active 的 `entry`，直接替换 `me_value`；

[2] 搜索失败，返回 Unused 或 Dummy 的 `entry`，完整设置 `me_key`、`me_hash` 和 `me_value`。

在 Python 中，对 `PyDictObject` 对象插入或设置元素有两种情况，如下面的代码所示：

```
d = {}
d[1] = 1
d[1] = 2
```

第二行 Python 代码是在 `PyDictObject` 对象中没有这个 `entry` 的情况下插入元素，第三行是在 `PyDictObject` 对象中已经有这个 `entry` 的情况下重新设置元素。可以看到，insertdict 完全可以适应这两种情况，在 `insertdict` 中，代码清单 5-5 的[2]处理第二行 Python 代码，代码清单 5-5 的[1]处理第三行 Python 代码。实际上，这两行 Python 代码也确实都调用了 `insertdict`。

当这两行设置 `PyDictObject` 对象元素的 Python 代码被 Python 虚拟机执行时，并不

是直接就调用 insertdict，因为观察代码可以看到，insertdict 需要一个 hash 值作为调用参数，那这个 hash 值是在什么地方获得的呢？实际上，在调用 insertdict 之前，还会调用 PyDict_SetItem（见代码清单 5-6）。

代码清单 5-6

```
[dictobject.c]
int PyDict_SetItem(register PyObject *op, PyObject *key, PyObject *value)
{
    register dictobject *mp;
    register long hash;
    register Py_ssize_t n_used;

    mp = (dictobject *)op;
    //[1]: 计算 hash 值
    if (PyString_CheckExact(key)) {
        hash = ((PyStringObject *)key)->ob_shash;
        if (hash == -1)
            hash = PyObject_Hash(key);
    }
    else {
        hash = PyObject_Hash(key);
        if (hash == -1)
            return -1;
    }
    //[2]: 插入(key, value)元素对
    n_used = mp->ma_used;
    insertdict(mp, key, hash, value);

    //[3]: 必要时调整 dict 的内存空间
    if (!(mp->ma_used > n_used && mp->ma_fill*3 >= (mp->ma_mask+1)*2))
        return 0;
    return dictresize(mp, mp->ma_used * (mp->ma_used>50000 ? 2 : 4));
}
```

在 PyDict_SetItem 中，会首先在代码清单 5-6 的[1]处获得 key 的 hash 值，在上面的例子中，也就是一个 PyIntObject 对象 1 的 hash 值。然后代码清单 5-6 的[2]处通过 insertdict 进行元素的插入或设置。

PyDict_SetItem 在插入或设置元素的动作结束之后，并不会草草返回了事。接下来，它会检查是否需要改变 PyDictObject 内部 ma_table 所维护的内存区域的大小，在以后的叙述中，我们将这块内存称为“table”。那么什么时候需要改变 table 的大小呢？在前面我们说过，如果 table 的装载率大于 2/3 时，后续的插入动作遭遇到冲突的可能性会非常大。所以装载率是否大于或等于 2/3 就是判断是否需要改变 table 大小的准则。在 PyDict_SetItem 中，有如下的代码：

```
    if (!(mp->ma_used > n_used && mp->ma_fill*3 >= (mp->ma_mask+1)*2))
        return 0;
```

经过转换，实际上可以得到：

```
(mp->ma_fill)/(mp->ma_mask+1) >= 2/3
```

这个等式左边的表达式正是装载率。然而装载率只是判定是否需要改变 table 大小的一个标准，还有另一个标准是在 insertdict 的过程中，是否使用了一个处于 Unused 态或 Dummy 态的 entry。前面我们说过，在搜索失败时，会返回一个 Dummy 态或 Unused 态的 entry，insertdict 会对这个 entry 进行填充。只有当这种情况发生并且装载率超标时，才会进行改变 table 大小的动作。而判断在 insertdict 的过程中是否填充了 Unused 态或 Dummy 态 entry，是通过下面的条件判断完成的：

```
mp->ma_used > n_used
```

其中的 n_used 就是进行 insertdict 操作之前的 mp->ma_used。通过观察 mp->ma_used 是否改变，就可以知道是否有 Unused 态或 Dummy 态的 entry 被填充。

在改变 table 时，并不一定是增加 table 的大小，同样也可能是减小 table 的大小。更改 table 的大小时，新的 table 的空间为：

```
mp->ma_used*(mp->ma_used>50000 ? 2 : 4)
```

如果一个 PyDictObject 对象的 table 中只有几个 entry 处于 Active 态，而大多数 entry 都处于 Dummy 态，那么改变 table 大小的结果显然就是减小了 table 的空间大小。

在确定新的 table 的大小时，通常选用的策略是时新的 table 中 entry 的数量是现在 table 中 Active 态 entry 数量的 4 倍，选用 4 倍是为了使 table 中处于 Active 态的 entry 的分布更加稀疏，减少插入元素时的冲突概率。当然，这是以内存空间为代价的。由于机器的内存是有限的，Python 总不能没心没肺地在任何时候都要求 4 倍空间，这样做，别的程序会有意见的☺。所以当 table 中 Active 态的 entry 数量非常大时，Python 只会要求 2 倍的空间，这次又是以执行速度来交换内存空间。Python 2.5 将这个“非常大”的标准划定在 50000。如此一来，各得其所，万事大吉。

至于具体改变 table 大小的重任，则交到了 dictresize 一人的肩上(见代码清单 5-7)。

**代码清单 5-7**

```
[dictobject.c]
static int dictresize(dictobject *mp, int minused)
{
    Py_ssize_t newsize;
    dictentry *oldtable, *newtable, *ep;
    Py_ssize_t i;
    int is_oldtable_malloced;
    dictentry small_copy[PyDict_MINSIZE];
    //[1]: 确定新的 table 的大小
    for(newsize = PyDict_MINSIZE; newsize <= minused && newsize > 0; newsize <<= 1)
        ;
    oldtable = mp->ma_table;
    is_oldtable_malloced = (oldtable != mp->ma_smalltable);

    //[2]: 新的 table 可以使用 mp->ma_smalltable
```

```
    if (newsize == PyDict_MINSIZE) {
        newtable = mp->ma_smalltable;
        if (newtable == oldtable) {
            if (mp->ma_fill == mp->ma_used) {
                //没有任何 Dummy 态 entry,直接返回
                return 0;
            }
            //将旧 table 拷贝,进行备份
            memcpy(small_copy, oldtable, sizeof(small_copy));
            oldtable = small_copy;
        }
    }
    //[3]: 新的 table 不能使用 mp->ma_smalltable,需要在系统堆上申请
    else {
        newtable = PyMem_NEW(dictentry, newsize);
    }

    //[4]: 设置新 table
    mp->ma_table = newtable;
    mp->ma_mask = newsize - 1;
    memset(newtable, 0, sizeof(dictentry) * newsize);
    mp->ma_used = 0;
    i = mp->ma_fill;
    mp->ma_fill = 0;

    //[5]: 处理旧 table 中的 entry:
    //      1、Active 态 entry,搬移到新 table 中
    //      2、Dummy 态 entry,调整 key 的引用计数,丢弃该 entry
    for (ep = oldtable; i > 0; ep++) {
        if (ep->me_value != NULL) { /* active entry */
            --i;
            insertdict(mp, ep->me_key, ep->me_hash, ep->me_value);
        }
        else if (ep->me_key != NULL) {  /* dummy entry */
            --i;
            assert(ep->me_key == dummy);
            Py_DECREF(ep->me_key);
        }
    }
    //[6]: 必要时释放旧 table 所维护的内存空间
    if (is_oldtable_malloced)
        PyMem_DEL(oldtable);
    return 0;
}
```

我们看一看在改变 `dict` 的内存空间时所发生的动作,如代码清单 5-7 中的[1]、[2]、[3]、[4]、[5]、[6]所示。

[1] `dictresize` 首先会确定新的 table 的大小,很显然,这个大小一定要大于传入的参数 `minused`,这个 `minused` 的确定我们在前面已经看到了,这是 Python 在调用 `dictresize` 时要求 `dictresize` 必须保证的内存空间,只许超出,不许偷工减料。`dictresize` 从 8 开始,以指数方式增加大小,直到超过了 `minused` 为

止。所以实际上新的 table 的大小在大多数情况下至少是原来 table 中 Active 态 entry 数量的 4 倍。

[2]、[3] 如果在代码清单 5-7 的[1]中获得的新的 table 大小为 8，则不需要在堆上分配空间，直接使用 ma_smalltable 就可以了；否则，则需要在堆上分配空间。

[4] 对新的 table 进行初始化，并调整原来 PyDictObject 对象中用于维护 table 使用情况的变量。

[5] 对原来 table 中的非 Unused 态 entry 进行处理。对于 Active 态 entry，显然需要将其插入到新的 table 中，这个动作由前面考察过的 insertdict 完成；而对于 Dummy 态的 entry，则将该 entry 丢弃，当然，要调整 entry 中 key 的引用计数。之所以能将 Dummy 态 entry 丢弃，是因为 Dummy 态 entry 存在的唯一理由就是为了不使搜索时的探测链中断。现在所有 Active 态的 entry 都重新依次插入新的 table 中，它们会形成一条新的探测序列，不再需要这些 Dummy 态的 entry 了。

[6] 如果之前旧的 table 指向了一片系统堆中的内存空间，那么我们还需要释放这片内存空间，防止内存泄露。

现在，利用我们对 PyDictObject 的认识，想象一下从 table 中删除一个元素应该怎样操作呢？

```
[dictobject.c]
int PyDict_DelItem(PyObject *op, PyObject *key)
{
    register dictobject *mp;
    register long hash;
    register dictentry *ep;
    PyObject *old_value, *old_key;
    //[1]: 获得 hash 值
    if (!PyString_CheckExact(key) ||
        (hash = ((PyStringObject *) key)->ob_shash) == -1) {
        hash = PyObject_Hash(key);
        if (hash == -1)
            return -1;
    }
    //[2]: 搜索 entry
    mp = (dictobject *)op;
    ep = (mp->ma_lookup)(mp, key, hash);
    if (ep->me_value == NULL) { //搜索失败，entry 不存在
        return -1;
    }
    //[3]: 删除 entry 所维护的元素，将 entry 的状态转为 dummy 态
    old_key = ep->me_key;
    ep->me_key = dummy;
```

```
    old_value = ep->me_value;
    ep->me_value = NULL;
    mp->ma_used--;
    Py_DECREF(old_value);
    Py_DECREF(old_key);
    return 0;
}
```

流程非常清晰，先计算 hash 值，然后搜索相应的 entry，最后删除 entry 中维护的元素，并将 entry 从 Active 态变换为 Dummy 态，同时还将调整 PyDictObject 对象中维护 table 使用情况的变量。

## 5.3.4 操作示例

下面我们用一个简单的例子来动态地展示对 PyDictObject 中 table 的维护过程，需要提醒的是，这里采用的散列函数和探测函数都与 Python 中 PyDictObject 实际采用的策略不同，这里只是从观念上展示对 table 的维护过程。在下面的图中，**白色背景**元素代表 Unused 态 entry，**灰色背景**元素为 Active 态，**交叉图饰**背景元素为 Dummy 态。

假如 table 中有 10 个 entry，散列函数为 HASH(x) = x mod 10，冲突解决方案采用线性探测，且探测函数为 x = x + 1。假设向 table 中依次加入了以下元素对：(4，4)，(14，14)，(24，24)，(34，34)，则加入元素后的 entry 的 dict 如图 5-5 所示：




图 5-5 插入与删除示例图之一

现在删除元素对（14，14），位置#5 处的 entry 将从 Active 态进入 Dummy 态。然后我们向 table 中插入新的元素对（104，104），则在搜索的过程中，由于原来位置#5 处维护 14 的 entry 现在处于 Dummy 态，所以 freeslots 会指向这个可用的 entry，如图 5-6 所示：

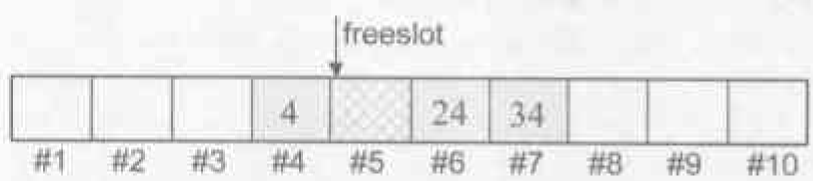



图 5-6 插入与删除示例图之二

搜索完成后，填充 freeslot 所指向的 entry，其结果如图 5-7 所示：




图 5-7 插入与删除示例图之三

然后，再向 table 中插入元素对（14，14），这时由于探测序列上已经没有 Dummy 态的 entry 了，所以最后返回的 ep 会指向一个处于 Unused 态的 entry，如图 5-8 所示：

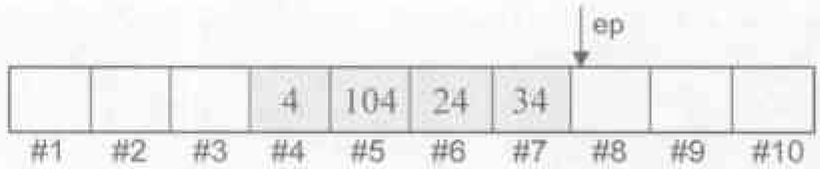



图 5-8 插入与删除示例图之四

最后插入元素对（14，14），结果如图 5-9 所示：

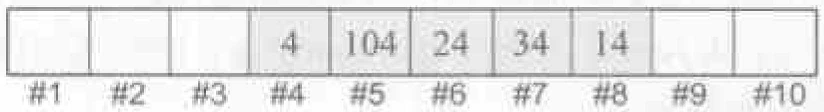



图 5-9 插入与删除示例图之五

# 5.4 PyDictObject 对象缓冲池

前面我们提到，在 PyDictObject 的实现机制中，同样使用了缓冲池的技术。现在，我们来看看 PyDictObject 对象的缓冲池：

```
[dictobject.c]
#define MAXFREEDICTS 80
static PyDictObject *free_dicts[MAXFREEDICTS];
static int num_free_dicts = 0;
```

实际上，PyDictObject 中使用的这个缓冲池机制与 PyListObject 中使用的缓冲池机制是一样的。开始时，这个缓冲池里什么都没有，直到第一个 PyDictObject 被销毁时，这个缓冲池才开始接纳被缓冲的 PyDictObject 对象（见代码清单 5-8）。

**代码清单 5-8**

```
[dictobject.c]
static void dict_dealloc(register dictobject *mp)
{
    register dictentry *ep;
    Py_ssize_t fill = mp->ma_fill;
    //[1]: 调整 dict 中对象的引用计数
    for (ep = mp->ma_table; fill > 0; ep++) {
        if (ep->me_key) {
            --fill;
            Py_DECREF(ep->me_key);
            Py_XDECREF(ep->me_value);
        }
    }
    //[2] : 释放从系统堆中申请的内存空间
    if (mp->ma_table != mp->ma_smalltable)
        PyMem_DEL(mp->ma_table);
    //[3] : 将被销毁的 PyDictObject 对象放入缓冲池
    if (num_free_dicts < MAXFREEDICTS && mp->ob_type == &PyDict_Type)
        free_dicts[num_free_dicts++] = mp;
    else
        mp->ob_type->tp_free((PyObject *)mp);
}
```

和 PyListObject 中缓冲池的机制一样，缓冲池中只保留了 PyDictObject 对象。如果 PyDictObject 对象中 ma_table 维护的是从系统堆申请的内存空间，那么 Python 将释放这块内存空间，归还给系统堆。而如果被销毁的 PyDictObject 中的 table 实际上并没有从系统堆中申请，而是指向 PyDictObject 固有的 ma_smalltable，那么只需要调整 ma_smalltable 中的对象引用计数就可以了。

在创建新的 PyDictObject 对象时，如果在缓冲池中有可以使用的对象，则直接从缓冲池中取出使用，而不需要再重新创建：

```
[dictobject.c]
PyObject* PyDict_New(void)
{
    register dictobject *mp;
    ............
    if (num_free_dicts) {
        mp = free_dicts[--num_free_dicts];
        _Py_NewReference((PyObject *)mp);
        if (mp->ma_fill) {
            EMPTY_TO_MINSIZE(mp);
        }
    }
    ............
}
```

## 5.5 Hack PyDictObject

现在我们可以根据对 PyDictObject 的了解，在 Python 源代码中添加代码，动态而真实地观察 Python 运行时 PyDictObject 的一举一动了。

我们首先来观察，在 insertdict 发生之后，PyDictObject 对象中 table 的变化情况。由于 Python 内部大量地使用 PyDictObject，所以对 insertdict 的调用会非常频繁，成千上万的 PyDictObject 对象会排着长队来依次使用 insertdict。如果只是简单地输出，我们立刻就会被淹没在输出信息中。所以我们需要一套机制来确保当 insertdict 发生在某一特定的 PyDictObject 对象身上时，才会输出信息。这个 PyDictObject 对象当然是我们自己创建的对象，必须使它有区别于 Python 内部使用的 PyDictObject 对象的特征。这个特征，在这里，我把它定义为 PyDictObject 包含“PR”的 PyStringObject 对象，当然，你也可以选用自己的特征串。如果在 PyDictObject 中找到了这个对象，则输出信息：

```
static void ShowDictObject(dictobject* dictObject)
{
    dictentry* entry = dictObject->ma_table;
    int count = dictObject->ma_mask+1;
    int i;
    //输出 key
```

```
    printf("  key : ");
    for(i = 0; i < count; ++i) {
        PyObject* key = entry->me_key;
        PyObject* value = entry->me_value;
        if(key == NULL) {
            printf("NULL");
        }
        else {
            if(PyString_Check(key)) {
                if(PyString_AsString(key)[0] == '<') {
                    printf("dummy");
                }
                else {
                    (key->ob_type)->tp_print(key, stdout, 0);
                }
            }
            else{
                (key->ob_type)->tp_print(key, stdout, 0);
            }
        }
        printf("\t");
        ++entry;
    }
    //输出 value
    printf("\nvalue : ");
    entry = dictObject->ma_table;
    for(i = 0; i < count; ++i) {
        PyObject* key = entry->me_key;
        PyObject* value = entry->me_value;
        if(value == NULL) {
            printf("NULL");
        }
        else {
            (key->ob_type)->tp_print(value, stdout, 0);
        }
        printf("\t");
        ++entry;
    }
    printf("\n");
}
static void
insertdict(register dictobject *mp, PyObject *key, long hash, PyObject
  *value)
{
    ……
    {
        dictentry *p;
        long strHash;
        PyObject* str = PyString_FromString("PR");
        strHash = PyObject_Hash(str);
        p = mp->ma_lookup(mp, str, strHash);
        if(p->me_value != NULL && (key->ob_type)->tp_name[0] == 'i') {
            PyIntObject* intObject = (PyIntObject*)key;
            printf("insert %d\n", intObject->ob_ival);
            ShowDictObject(mp);
        }
```

```
    }
}
```

对于 `PyDictObject` 对象，依次插入 9 和 17，根据 `PyDictObject` 选用的 hash 策略，这两个数会产生冲突，9 的 hash 结果为 1，而 17 经过再次探测后，会获得 hash 结果为 7。图 5-10 中的前两个结果显示了这个过程。

```
>>> d = {'PR':1 }
>>> d[9] = 9
insert 9
  key : 'PR'    9       NULL    NULL    NULL    NULL    NULL    NULL
value : '1'     9       NULL    NULL    NULL    NULL    NULL    NULL
>>> d[17] = 17
insert 17
  key : 'PR'    9       NULL    NULL    NULL    NULL    NULL    17
value : '1'     9       NULL    NULL    NULL    NULL    NULL    17
>>> del d[9]
>>> d[17] = 16
insert 17
  key : 'PR'    dummy   NULL    NULL    NULL    NULL    NULL    17
value : '1'     NULL    NULL    NULL    NULL    NULL    NULL    16
>>> del d[17]
>>> d[17] = 17
insert 17
  key : 'PR'    17      NULL    NULL    NULL    NULL    NULL    dummy
value : '1'     17      NULL    NULL    NULL    NULL    NULL    NULL
```

图 5-10 dict 变动时 table 的变化情况

然后我们将 9 删除，则原来 9 的位置会出现一个 `dummy` 态的标识。接着将 17 删除，并再次插入 17，显然，17 应该出现在原来 9 的位置，而原来 17 的位置则是 `dummy` 标识。图 5-10 中的后两个结果显示了这个过程。

下面我们观察 Python 自身对 `PyDictObject` 的使用情况，在 `dict_dealloc` 中添加代码监控 Python 在执行时调用 `dict_dealloc` 的频度，图 5-11 是监测结果：

```
>>> i = 1
dealloc dict : size 1   num_free_dicts is : 1
dealloc dict : size 1   num_free_dicts is : 2
dealloc dict : size 0   num_free_dicts is : 3
dealloc dict : size 0   num_free_dicts is : 4
dealloc dict : size 0   num_free_dicts is : 5
dealloc dict : size 0   num_free_dicts is : 6
dealloc dict : size 0   num_free_dicts is : 7
dealloc dict : size 2   num_free_dicts is : 3
dealloc dict : size 1   num_free_dicts is : 4
dealloc dict : size 0   num_free_dicts is : 5
dealloc dict : size 0   num_free_dicts is : 6
dealloc dict : size 0   num_free_dicts is : 7
dealloc dict : size 1   num_free_dicts is : 8
dealloc dict : size 1   num_free_dicts is : 9
>>> print 'hello world'
dealloc dict : size 0   num_free_dicts is : 1
dealloc dict : size 0   num_free_dicts is : 2
dealloc dict : size 0   num_free_dicts is : 3
```

图 5-11 Python 执行时使用 dict 的情况

我们前面已经说了，Python 内部大量使用了 PyDictObject 对象，然而监测的结果还是让我们惊讶不已，原来对于一个简简单单的赋值，一个简简单单的打印，Python 内部都会创建并销毁多达 14 个的 PyDictObject 对象。不过这 14 个 dict 中有一些是重新申请的，所以最终缓冲池中的自由 dict 对象总是 9 个。这 14 个 dict 对象中有一些参与编译过程的 PyDictObject 对象，所以在执行一个完整的 Python 源文件时，并不是每一行都会销毁如此庞大的 dict 群。当然，我们可以看到，这些 PyDictObject 对象中 entry 的个数都很少，所以只需要使用 ma_smalltable 就可以了。这里，也提醒我们 PyDictObject 缓冲池的重要性。

所以我们也监控了缓冲池的使用，在 dict_print 中添加代码，打印当前的 num_free_dicts 值。监控结果见图 5-12。有一点奇怪的是，在创建了 d2 和 d3 之后，num_free_dicts 的值仍然都是 9。直觉上来讲，它们对应的是应该是 6 和 5 才对。但是，看一看图 5-11，其实在执行 print 语句的时候，同“i = 1”这样的赋值语句一样，同样会调用 dealloc 操作 14 次，最后也同样会剩下 9 个自由的 dict 对象，所以每次打印出来，num_free_dicts 的值都是 9。后来 del d1 和 del d2 时，每次除了 Python 虚拟机例行的 14 次销毁外，还有我们自己创建的 dict 对象的销毁，所以打印出来的 num_free_dicts 的值变为了 10 和 11。

```
>>> d1 = {}
>>> print d1
num_free_dicts is : 9

>>> d2 = {}
>>> print d2
num_free_dicts is : 9

>>> d3 = {}
>>> print d3
num_free_dicts is : 9

>>> del d1
>>> print d3
num_free_dicts is : 10

>>> del d2
>>> print d3
num_free_dicts is : 11
```

图 5-12 监控 dict 缓冲池

CHAPTER

第 6 章

# 最简单的Python模拟——Small Python

## 6.1 Small Python

在详细考察了 Python 中最常用的几个对象之后，我们现在完全可以利用这些对象做出一个最简单的 Python。这一章的目的就是模拟出这样一个最简单的 Python，我们把它称为 Small Python。

在 Small Python 中，我们首先需要实现之前已经分析过的那些对象，比如 `PyIntObject`，与 CPython 不同的是，我们并没有实现像 CPython 那样复杂的机制，作为一个模拟程序，我们只实现了简单的功能，也没有引入对象缓冲池的机制。这一切都是为了简洁而清晰地展示出 Python 运行时的脉络。

在 Small Python 中，实际上还需要实现 Python 的运行时环境，Python 的运行时环境是我们在以后的章节中将要剖析的重点。在这里只是展示了其核心的思想——利用 `PyDictObject` 对象来维护变量名到变量值的映射。

当然，在 CPython 中，还有许多其他的主题，比如 Python 源代码的编译，Python 字节码的生成和执行等，在 Small Python 中，我们都不会涉及，因为到目前为止，我们没有任何能力做出如此逼真的模拟。不过当我们跟随本书完成了 Python 源代码的探索之后，就完全有能力实现一个真正的 Python 了。

在 Small Python 中，我们仅仅实现了 `PyIntObject`、`PyStringObject` 及 `PyDictObject` 对象，仅仅实现了加法运算和输出操作。同时编译的过程也被简化到了极致，因此我们的 Small Python 只能处理非常受限的表达式。虽然很简陋，但从中可以看到 Python 的骨架，同时，这也是我们深入 Python 虚拟机和运行时的起点。

## 6.2 对象机制

在 Small Python 中，对象机制与 CPython 完全相同：

```
[PyObject]
#define PyObject_HEAD \
    int refCount;\
    struct tagPyTypeObject *type

#define PyObject_HEAD_INIT(typePtr)\
    0, typePtr

typedef struct tagPyObject
{
    PyObject_HEAD;
}PyObject;
```

但是对于类型对象，我们进行了大规模的删减。最终在类型对象中，只定义了加法操作，hash 操作以及输出操作：

```
[PyTypeObject]
typedef void (*PrintFun)(PyObject* object);
typedef PyObject* (*AddFun)(PyObject* left, PyObject* right);
typedef long (*HashFun)(PyObject* object);

typedef struct tagPyTypeObject
{
    PyObject_HEAD;
    char* name;
    PrintFun print;
    AddFun add;
    HashFun hash;
}PyTypeObject;
```

`PyIntObject` 的实现与 CPython 几乎是一样的，不过没有复杂的对象缓冲机制：

```
[PyIntObject]
typedef struct tagPyIntObject
{
    PyObject_HEAD;
    int value;
}PyIntObject;

PyObject* PyInt_Create(int value)
{
    PyIntObject* object = new PyIntObject;
    object->refCount = 1;
    object->type = &PyInt_Type;
    object->value = value;
    return (PyObject*)object;
}

static void int_print(PyObject* object)
{
```

```
    PyIntObject* intObject = (PyIntObject*)object;
    printf("%d\n", intObject->value);
}

static PyObject* int_add(PyObject* left, PyObject* right)
{
    PyIntObject* leftInt = (PyIntObject*)left;
    PyIntObject* rightInt = (PyIntObject*)right;
    PyIntObject* result = (PyIntObject*)PyInt_Create(0);
    if(result == NULL)
    {
        printf("We have no enough memory!!");
        exit(1);
    }
    else
    {
        result->value = leftInt->value + rightInt->value;
    }
    return (PyObject*)result;
}

static long int_hash(PyObject* object)
{
    return (long)((PyIntObject*)object)->value;
}

PyTypeObject PyInt_Type =
{
    PyObject_HEAD_INIT(&PyType_Type),
    "int",
    int_print,
    int_add,
    int_hash
};
```

Small Python 中的 `PyStringObject` 与 CPython 中大不相同，在 CPython 中，`PyStringObject` 是一个变长对象，而 Small Python 中只是一个简单的定长对象，因为 Small Python 的定位就是个演示的程序：

```
[PyStrObject]
typedef struct tagPyStrObject
{
    PyObject_HEAD;
    int length;
    long hashValue;
    char value[50];
}PyStringObject;

PyObject* PyStr_Create(const char* value)
{
    PyStringObject* object = new PyStringObject;
    object->refCount = 1;
    object->type = &PyString_Type;
    object->length = (value == NULL) ? 0 : strlen(value);
    object->hashValue = -1;
```

```
    memset(object->value, 0, 50);
    if(value != NULL)
    {
        strcpy(object->value, value);
    }
    return (PyObject*)object;
}

static void string_print(PyObject* object)
{
    PyStringObject* strObject = (PyStringObject*)object;
    printf("%s\n", strObject->value);
}

static long string_hash(PyObject* object)
{
    PyStringObject* strObject = (PyStringObject*)object;
    register int len;
    register unsigned char *p;
    register long x;

    if (strObject->hashValue != -1)
        return strObject->hashValue;
    len = strObject->length;
    p = (unsigned char *)strObject->value;
    x = *p << 7;
    while (--len >= 0)
        x = (1000003*x) ^ *p++;
    x ^= strObject->length;
    if (x == -1)
        x = -2;
    strObject->hashValue = x;
    return x;
}

static PyObject* string_add(PyObject* left, PyObject* right)
{
    PyStringObject* leftStr = (PyStringObject*)left;
    PyStringObject* rightStr = (PyStringObject*)right;
    PyStringObject* result = (PyStringObject*)PyStr_Create(NULL);
    if(result == NULL)
    {
        printf("We have no enough memory!!");
        exit(1);
    }
    else
    {
        strcpy(result->value, leftStr->value);
        strcat(result->value, rightStr->value);
    }
    return (PyObject*)result;
}

PyTypeObject PyString_Type =
{
    PyObject_HEAD_INIT(&PyType_Type),
```

```
    "str",
    string_print,
    string_add,
    string_hash
};
```

在 Python 的解释器工作时，还有一个非常重要的对象，PyDictObject 对象。PyDictObject 对象在 Python 运行时会维护变量名和变量值的映射关系，Python 所有的动作都是基于这种映射关系的。在 Small Python 中，我们基于 C++中的 map 来实现 PyDictObject 对象。当然，map 的运行效率比 CPython 中所采用的 hash 技术会慢一些，而且，对于散列冲突的情况，map 也没有办法解决，但是对于我们的 Small Python，map 就足够了：

```
[PyDictObject]
typedef struct tagPyDictObject
{
    PyObject_HEAD;
    map<long, PyObject*> dict;
}PyDictObject;

PyObject* PyDict_Create()
{
    PyDictObject* object = new PyDictObject;
    object->refCount = 1;
    object->type = &PyDict_Type;

    return (PyObject*)object;
}

PyObject* PyDict_GetItem(PyObject* target, PyObject* key)
{
    long keyHashValue = (key->type)->hash(key);
    map<long, PyObject*>& dict = ((PyDictObject*)target)->dict;
    map<long, PyObject*>::iterator it = dict.find(keyHashValue);
    map<long, PyObject*>::iterator end = dict.end();
    if(it == end)
    {
        return NULL;
    }
    return it->second;
}

int PyDict_SetItem(PyObject* target, PyObject* key, PyObject* value)
{
    long keyHashValue = (key->type)->hash(key);
    PyDictObject* dictObject = (PyDictObject*)target;
    (dictObject->dict)[keyHashValue] = value;
    return 0;
}

//function for PyDict_Type
static void dict_print(PyObject* object)
```

```
{
    PyDictObject* dictObject = (PyDictObject*)object;
    printf("{");
    map<long, PyObject*>::iterator it = (dictObject->dict).begin();
    map<long, PyObject*>::iterator end = (dictObject->dict).end();
    for( ; it != end; ++it)
    {
        //print key
        printf("%ld : ", it->first);
        //print value
        PyObject* value = it->second;
        (value->type)->print(value);
        printf(", ");
    }
    printf("}\n");
}

PyTypeObject PyDict_Type =
{
    PyObject_HEAD_INIT(&PyType_Type),
        "dict",
        dict_print,
        0,
        0
};
```

Small Python 中的对象机制的所有内容都在上边列出了，非常简单，对吧，这就对了，要的就是这个简单☺。

## 6.3 解释过程

说 Small Python 中没有编译，对的，它根本就不会进行任何常规的编译动作，没有 token 解析，没有抽象语法树的建立。但说 Small Python 中有那么一点点编译的味道，其实也不错，我们叫这种动作为解释。无论如何，它至少要解析输入的语句，以判断这条语句到底是要干什么，它是要上山打虎呢，还是要下河摸鱼？如果连这最基本的都做不到，Small Python 还不如回家卖红薯得了。

然而 Small Python 中的这种解释动作还是被简化到了极致，它实际上就是简单的字符串查找加 if…else…结构：

```
void ExcuteCommand(string& command)
{
    string::size_type pos = 0;
    if((pos = command.find("print ")) != string::npos)
    {
        ExcutePrint(command.substr(6));
    }
    else if((pos = command.find(" = ")) != string::npos)
    {
```

```
        string target = command.substr(0, pos);
        string source = command.substr(pos+3);
        ExcuteAdd(target, source);
    }
}

void ExcuteAdd(string& target, string& source)
{
    string::size_type pos;
    PyObject* strValue = NULL;
    PyObject* resultValue = NULL;
    if(IsSourceAllDigit(source))
    {
        PyObject* intValue = PyInt_Create(atoi(source.c_str()));
        PyObject* key = PyStr_Create(target.c_str());
        PyDict_SetItem(m_LocalEnvironment, key, intValue);
    }
    else if(source.find("\"") != string::npos)
    {
        strValue = PyStr_Create(source.substr(1, source.size()-2).c_str());
        PyObject* key = PyStr_Create(target.c_str());
        PyDict_SetItem(m_LocalEnvironment, key, strValue);
    }
    else if((pos = source.find("+")) != string::npos)
    {
        PyObject* leftObject = GetObjectBySymbol(source.substr(0, pos));
        PyObject* rightObject =
          GetObjectBySymbol(source.substr(pos+1));
        if(leftObject != NULL && right != NULL
          && leftObject->type == rightObject->type)
        {
          resultValue = (leftObject->type)->add(leftObject,
            rightObject);
          PyObject* key = PyStr_Create(target.c_str());
          PyDict_SetItem(m_LocalEnvironment, key, resultValue);
        }
        (m_LocalEnvironment->type)->print(m_LocalEnvironment);
    }
}
```

通过字符串搜索，如果命令中出现“=”，就是一个赋值或加法过程；如果命令中出现“print”，就是一个输出过程。在 ExcuteAdd 中，还需要进行进一步地字符串搜索，以确定是否需要有一个额外的加法过程。根据这些解析的结果进行不同的动作，就是 Small Python 中的解释过程。这个过程在 CPython 中是通过正常的编译过程来实现的，而且最后会得到字节码的编译结果。

在这里需要重点指出的是那个 m_LocalEnvironment，这是一个 PyDictObject 对象，其维护着 Small Python 运行过程中，动态创建的变量的变量名和变量值的映射。这个就是 Small Python 中的执行环境，Small Python 正是靠它来维护运行过程中的所有变量的状态。在 CPython 中，运行环境实际上也是这样一个机制。当需要访问变量时，就从这个

PyDictObject 对象中查找变量的值。这一点在执行输出操作时可以看得很清楚：

```
PyObject* GetObjectBySymbol(string& symbol)
{
    PyObject* key = PyStr_Create(symbol.c_str());
    PyObject* value = PyDict_GetItem(m_LocalEnvironment, key);
    if(value == NULL)
    {
        cout << "[Error] : " << symbol << " is not defined!!" << endl;
        return NULL;
    }
    return value;
}


void ExcutePrint(string symbol)
{
    PyObject* object = GetObjectFromSymbol(symbol);
    if(object != NULL)
    {
        PyTypeObject* type = object->type;
        type->print(object);
    }
}
```

在这里，通过变量名 symbol，获得了变量值 object。而在刚才的 ExcueteAdd 中，我们将变量名和变量值建立了联系，并存放到 m_LocalEnvironment 中。这种一进一出的机制正是 CPython 执行时的关键，在以后对 Python 字节码解释器的详细剖析中，我们将真实而具体地看到这种机制。

## 6.4 交互式环境

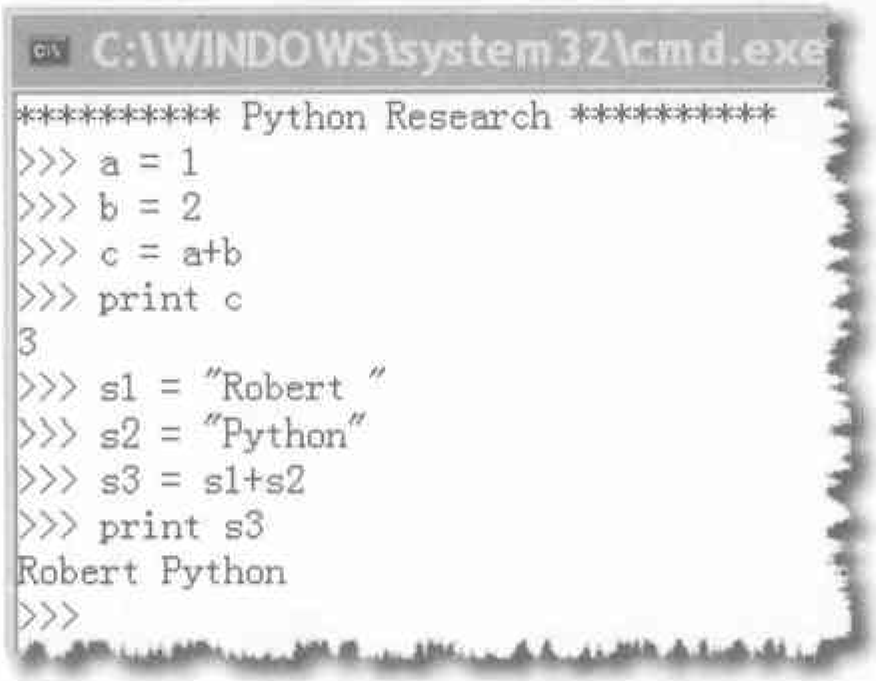
好了，我们的 Small Python 几乎已经完成了，最后所缺的就是一个交互式环境：

```
const char* info = "********** Python Research **********\n";
const char* prompt = ">>> ";

void Excute()
{
    cout << info;
    cout << prompt;
    while(getline(cin, m_Command)){
        if(m_Command.size() == 0){
            cout << prompt;
            continue;
        }
        else if(m_Command == "exit"){
            return;
        }
        else{
            ExcuteCommand(m_Command);
```

```
        }
        cout << prompt;
    }
}
```

有了它，我们的 Small Python 就大功告成了。现在，来看一看我们的成果吧：




到这里，我们就结束了对 Python 中内建对象的探索，通过 Small Python 这一个简陋的承前启后的东西，我们将敲开 Python 虚拟机的大门。那里是字节码、虚拟机、条件判断语句、函数、类、异常等的神秘世界。好了，打起精神，我们出发了……

# 第 2 部分

# Python虚拟机

CHAPTER

第 7 章

# Python 的编译结果——Code 对象与 pyc 文件

平时我们可能每天都会写一些这样或那样的 Python 程序，或者是处理文本，或者是进行系统管理工作。要运行这些程序，我们或者是双击文件图标（Windows 平台），或者是在命令行环境下键入`python my-program.py`这样的命令（Windows 平台或 Linux 平台），只要完成这些动作，Python 程序就如我们所预期的那样开始工作，那么，一个文本方式的.py 文件是怎样转换成一系列的机器指令并被执行的呢？

## 7.1 Python 程序的执行过程

事实上，.py 文件中的 Python 语句并没有被转换成一系列的机器指令。这一点可能我们都有所耳闻，因为坊间广泛地流传着这样一种说法：Python 是一种解释性的语言。这种说法是不正确的，实际上，尽管 Python 不如 Java，C#一样出身名门，但是 Python 的本质与 Java 和 C#是一样的，Python 程序的执行原理和 Java 程序、C#程序的执行原理都可以用两个词囊括——虚拟机、字节码。

我们知道，Python 有一个非常核心的东西，这个东西通常被称为解释器（interpreter），当我们在命令行下敲入 python 时，目的就是为了激活这个解释器。当我们通过`python my-program.py`执行一个特定的 Python 程序时，Python 解释器立即被激活，然后执行 Python 程序。在真正开始执行之前，实际上，Python 的解释器还要完成一个非常复杂的工作——编译.py 文件。

编译？没错，确实是编译。实际上，Python 解释器在执行任何一个 Python 程序文件时，首先进行的动作都是先对文件中的 Python 源代码进行编译，编译的主要结果是产生

一组 Python 的 byte code（字节码），然后将编译的结果交给 Python 的虚拟机（Virtual Machine），由虚拟机按照顺序一条一条地执行字节码，从而完成对 Python 程序的执行动作。那么这些 Python 的编译器还有 Python 的虚拟机在什么地方呢？难道在 python.exe 中？如果你仔细观察过 python.exe，你会发现这个可执行程序仅仅只有 24KB，不太可能容纳下一个编译器外带一个虚拟机吧。呃，别着急，我们先通过图 7-1 来看一看 Python 程序的执行过程：

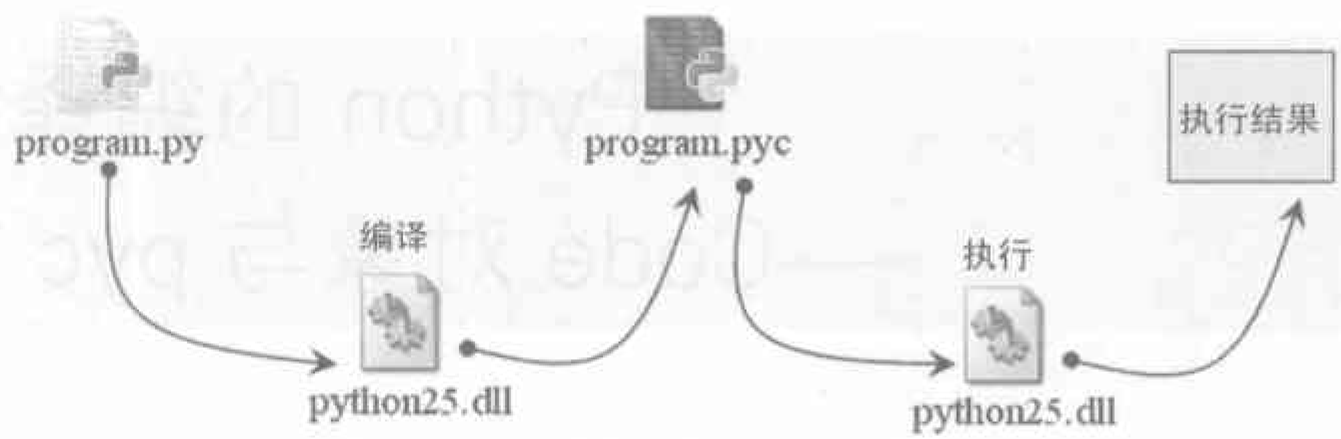



图 7-1 Python 程序的执行过程

细心的你一定发现了，我在编译和执行的地方都放上了一个 python25.dll，似乎 python25.dll 既当爹，又当妈，既完成了编译器的工作，又完成了虚拟机的工作。没错，Python 的编译器和虚拟机都藏身于这个 python 25.dll 中，在安装 Python 的时候，安装程序会自动将这个 dll 置于%systemroot%\system32（通常%systemroot%是 C:\WINDOWS）目录下。

对比一下 Java 程序，首先我们会有一个 program.java 的 Java 源程序，然后，用 javac 对 program.java 进行编译，产生一个 program.class 文件，最后调用 java 命令执行 program.class 中包含的 Java 字节码，得到结果。可以看到，program.py 对应 program.java，program.pyc 对应 program.class，这个过程与 Python 执行程序的过程实际是完全一致的。

虽然 Python 程序执行的机理与 Java 程序和 C#程序的执行机理是一样的，但是 Python 的虚拟机与 Java 和.NET 虚拟机还有不同之处。一个最大的不同是，Python 的虚拟机是一种更高级的虚拟机。这里的高级不是说 Python 的虚拟机的功能比 Java 和.NET 虚拟机的功能更强大、更拽，而是说与 Java 或.NET 相比，Python 的虚拟机距离真实机器更远。或者可以这么说，Python 虚拟机是一种抽象层次更高的虚拟机。

# 7.2 Python 编译器的编译结果——PyCodeObject 对象

## 7.2.1 PyCodeObject 对象与 pyc 文件

在图 7-1 中可以看到，Python 对源程序的编译结果是生成了一个.pyc 文件，然而这实际上是不太准确的。下面我们先通过考察一个简单的 Python 程序源文件来猜测一下，一个源文件被 Python 编译器编译之后究竟应该产生一些什么结果。

```
[demo.py]
class A:
    pass

def Fun():
    pass

a = A()
Fun()
```

现在我们已经知道，Python 在执行 demo.py 时，首先需要进行的动作就是对其进行编译，编译的结果是什么呢？当然有字节码，否则接下来的 Python 虚拟机也就没办法再玩下去了。然而除了字节码之外，编译的结果中还应包含其他一些信息，这些信息也是 Python 运行的时候所必需的。

看一下 demo.py，让我们充当 Python 编译器，用肉眼来解析一下，从这个文件中，我们可以看到，其中包含了一些字符串，一些常量值，还有一些操作。当然，Python 对操作的编译结果就是字节码。那么 Python 的编译过程对字符串和常量值的处理结果是什么呢？

在编译过程中，这些包含在 Python 源代码中的静态信息都会被 Python 编译器收集起来，编译的结果中包含了字符串，常量值，字节码等在源代码中出现的一切有用的静态信息。在 Python 运行期间，这些源文件中提供的静态信息最终会被存储在一个运行时的对象中，当 Python 运行结束后，这个运行时对象中所包含的信息甚至还会被存储在一种文件中。这个对象和文件就是我们这章探索的重点：PyCodeObject 对象和 pyc 文件。

从上面的描述中，可以看出，尽管图 7-1 中编译的结果是一个 pyc 文件，但是在 pyc 文件中，正襟危坐的其实是一个 PyCodeObject 对象，对于 Python 编译器来说，PyCodeObject 对象才是其真正的编译结果，而 pyc 文件只是这个对象在硬盘上的表现形式，它们实际上是 Python 对源文件编译的结果的两种不同存在方式。

在程序运行期间，编译结果存在于内存的 PyCodeObject 对象中；而 Python 结束运行后，编译结果又被保存到了 pyc 文件中。当下一次运行相同的程序时，Python 会根据 pyc

文件中记录的编译结果直接建立内存中的 PyCodeObject 对象，而不用再次对源文件进行编译了。

## 7.2.2 Python 源码中的 PyCodeObject

对 Python 的编译过程，我们不做过多的剖析，毕竟，Python 的编译过程和编译原理中描述的过程没有太多的区别。我们把关注的重点放在 Python 的编译结果上，要彻底地理解 Python 虚拟机的运行行为，必须要彻底地理解 Python 的编译结果——PyCodeObject。

先来看一看在 Python 源码中对 PyCodeObject 的声明：

```
[code.h]
typedef struct {
    PyObject_HEAD
    int co_argcount;        /* #arguments, except *args */
    int co_nlocals;      /* #local variables */
    int co_stacksize;       /* #entries needed for evaluation stack */
    int co_flags;       /* CO_..., see below */
    PyObject *co_code;      /* instruction opcodes */
    PyObject *co_consts;    /* list (constants used) */
    PyObject *co_names;     /* list of strings (names used) */
    PyObject *co_varnames;  /* tuple of strings (local variable names) */
    PyObject *co_freevars;  /* tuple of strings (free variable names) */
    PyObject *co_cellvars;      /* tuple of strings (cell variable names) */
    /* The rest doesn't count for hash/cmp */
    PyObject *co_filename;  /* string (where it was loaded from) */
    PyObject *co_name;      /* string (name, for reference) */
    int co_firstlineno;     /* first source line number */
    PyObject *co_lnotab;    /* string (encoding addr<->lineno mapping) */
    void *co_zombieframe /*for optimization only */
} PyCodeObject;
```

PyCodeObject 对象中的各个域各包含了什么信息，我们现在可以暂时不理会，在以后的剖析中，我们会一步一步将 PyCodeObject 的各个域里都包含了哪些信息全挖掘出来。这里可以稍稍透露一下，在 co_code 域存放的就是编译所生成的字节码指令序列。

Python 编译器在对 Python 源代码进行编译的时候，对于代码中的一个 Code Block，会创建一个 PyCodeObject 对象与这段代码对应。那么如何确定多少代码算是一个 Code Block 呢？事实上，Python 有一个简单而清晰的规则：当进入一个新的名字空间，或者说作用域时，我们就算是进入了一个新的 Code Block 了。

回顾一下上一节的 demo.py 文件，在 Python 编译器对源代码完成编译之后，总共会创建 3 个 PyCodeObject 对象，一个是对应 demo.py 整个文件的，一个是对应 class A 所代表的 Code Block，而最后一个是对应 def Fun 所代表的 Code Blcok。

在这里，我们开始提及 Python 中一个至关重要的概念——名字空间。名字空间是符号的上下文环境，符号的含义取决于名字空间。更具体地说，一个变量名对应的变量值是什么，在 Python 中，这并不是确定的，而是需要通过名字空间来决定。

对于某个符号，比如说 a，在某个名字空间中，它可能是一个 `PyIntObject` 对象；而在另一个名字空间中，它则可能是一个 `PyStringObject` 对象。在一个名字空间中，一个符号只可能有一种意义，a 要么是 `PyIntObject`，要么是 `PyStringObject`。名字空间可以一个套一个地形成一条**名字空间链**，Python 虚拟机在执行的过程中，会有很大一部分时间消耗在从这条名字空间链中确定一个符号所对应的对象是什么。

如果你现在对名字空间的概念还不是太明了，不要紧，随着对 Python 源码剖析的深入，你一定会对名字空间以及 Python 在名字空间链上的行为有越来越深刻的理解。现在我们需要记住的是，一个我们前面所说的 Code Block，就对应着一个名字空间，它会对应一个 `PyCodeObject` 对象。在 Python 中，类、函数、module 都对应着一个独立的名字空间。因此，都会有一个 `PyCodeObject` 对象与其对应。所以，demo.py 经过 Python 编译器的编译后，一共得到 3 个 `PyCodeObject` 对象。

## 7.2.3 pyc 文件

每一个 `PyCodeObject` 对象中都包含了每一个 Code Block 中所有 Python 源代码经过编译后得到的 byte code 序列。前面我们提到，Python 会将这些字节码序列和 `PyCodeObject` 对象一起存储在 pyc 文件中。但不幸的是，事实并不总是这样。试着在命令行下执行一下 `python demo.py`，你会发现 Python 并没有产生一个对应的 pyc 文件。为什么呢？真实的原因不得而知，不过我们可以做出一个合理的猜测：有一些 Python 程序只是临时完成一些琐碎的工作，比如统计某个特定文件中的词频信息，这样的程序可能仅仅运行一次，然后就再也没用了，所以也就没有保存其对应的 pyc 文件的必要，因此，对于直接用 `python demo.py` 这样的形式执行的程序，Python 就不会存储编译结果了。

但是假如说 demo.py 中实现的是一个需要被重用的类时，我们就希望能存储其对应的 `PyCodeObject` 对象，这样下次 Python 就不会再次进行编译了。当在另外一个程序，比如说在 demo.py 中对 demo.py 进行一个 `import demo` 的动态加载动作之后，你就会发现，Python 会为其产生 pyc 文件了。

这意味着 Python 的 import 机制会触发 pyc 文件的生成。实际上，在 Python 运行的过程中，如果碰到 `import abc` 这样的语句，那么 Python 将到设定好的 path 中寻找 abc.pyc 或 abc.dll 文件，如果没有这些文件，而只是发现了 abc.py，那么 Python 会首先将 abc.py 编译成相应的 `PyCodeObject` 的中间结果，然后创建 abc.pyc 文件，并将中间结果写入该

文件。接下来，Python 才会对 abc.pyc 文件进行 `import` 的动作，实际上也就是将 abc.pyc 文件中的 `PyCodeObject` 重新在内存中复制出来。

关于 Python 的 import 机制，在后面的章节中将专门予以剖析，这里我们只简单地利用 Python 内建的 imp module 来完成对生成 pyc 文件的触发。当然，为了得到一个 py 文件对应的 pyc 文件有很多种方法，比如利用 Python 标准库中的 py_compile、compiler 工具。这些方法间没有什么优劣之分，不过我们在这里选用 import 机制罢了。很容易利用下面所示的 generator.py 来创建上面那段代码（CodeObjectt.py）对应的 pyc 文件。

```
[pyc_generator.py]
import imp
import sys

def generate_pyc(name):
    fp, pathname, description = imp.find_module(name)
    try:
        imp.load_module(name, fp, pathname, description)
    finally:
        if fp:
            fp.close()

if __name__ == '__main__':
    generate_pyc(sys.argv[1])
```

图 7-2 所示的是 demo.py 所对应的 demo.pyc 文件：

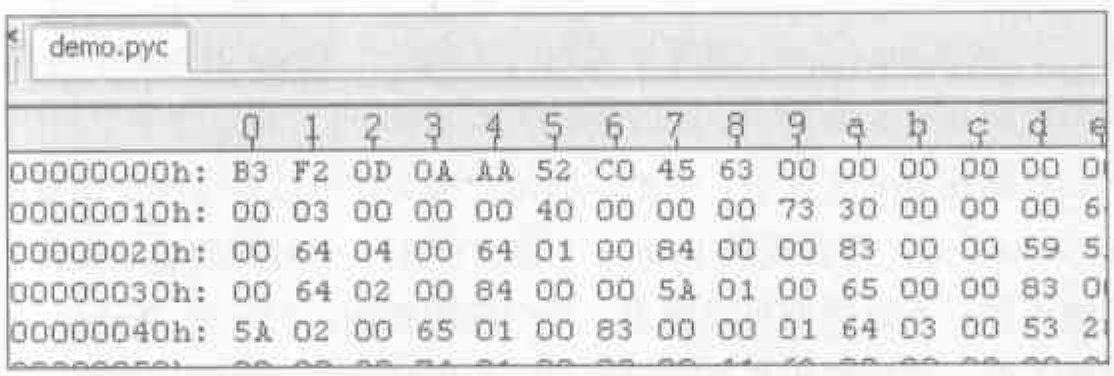



图 7-2 demo.pyc 的内容

可以看到，pyc 是一个二进制文件，那么 Python 如何解释这一堆看上去毫无意义的字节流就至关重要了。这也就是我们所关心的 pyc 文件的格式。

要了解 pyc 文件的格式，首先我们必须要清楚 `PyCodeObject` 中每一个域都表示什么含义，这一点是无论如何不能绕过去的。表 7-1 列出了 `PyCodeObject` 对象中各个域的具体意义。

表 7-1 PyCodeObject 中各个域的意义

| Field | Content |
|---|---|
| co_argcount | Code Block 的位置参数的个数，比如说一个函数的位置参数个数（位置参数请参见第 11 章对函数机制的剖析） |

续表

| Field | Content |
| --- | --- |
| co_nlocals | Code Block 中局部变量的个数，包括其位置参数的个数 |
| co_stacksize | 执行该段 Code Block 需要的栈空间 |
| co_flags | N/A |
| co_code | Code Block 编译所得的字节码指令序列。以 PyStringObject 的形式存在 |
| co_consts | PyTupleObject 对象，保存 Code Block 中的所有常量 |
| co_names | PyTupleObject 对象，保存 Code Block 中的所有符号 |
| co_varnames | Code Block 中的局部变量名集合 |
| co_freevars | Python 实现闭包需要用到的东西，后面的章节中会涉及 |
| co_cellvars | Code Block 中内部嵌套函数所引用的局部变量名集合 |
| co_filename | Code Block 所对应的.py 文件的完整路径 |
| co_name | Code Block 的名字，通常是函数名或类名 |
| co_firstlineno | Code Block 在对应的.py 文件中的起始行 |
| co_lnotab | 字节码指令与.py 文件中 source code 行号的对应关系，以 PyStringObject 的形式存在 |

标注为 N/A 的域是说明该域对于理解 Python 虚拟机的行为没有太多用处，所以我们在以后的剖析中也不会涉及。

这里需要说明一下的是 `co_lnotab` 域。在 Python 2.3 以前，有一条字节码指令，叫做 `SET_LINENO`，这条字节码会记录.py 文件中 source code 的位置（行号）信息，这个信息对于调试和显示异常信息都有用。但是，在 Python 2.3 之后，Python 在编译时不会再产生这条字节码，因为毕竟字节码代表的是运行时的行为，而记录源代码行号的动作完全可以在编译时完成。所以，相应地，Python 会在编译时直接将这个信息记录到 `co_lnotab` 中。

`co_lnotab` 中的字节码和相应 source code 行号的对应信息是以 unsigned bytes 的数组形式存在的，数组的形式可以看作（字节码指令在 co_code 中位置，source code 行号）形式的一个 `list`。比如对于下面的例子（见表 7-2）：

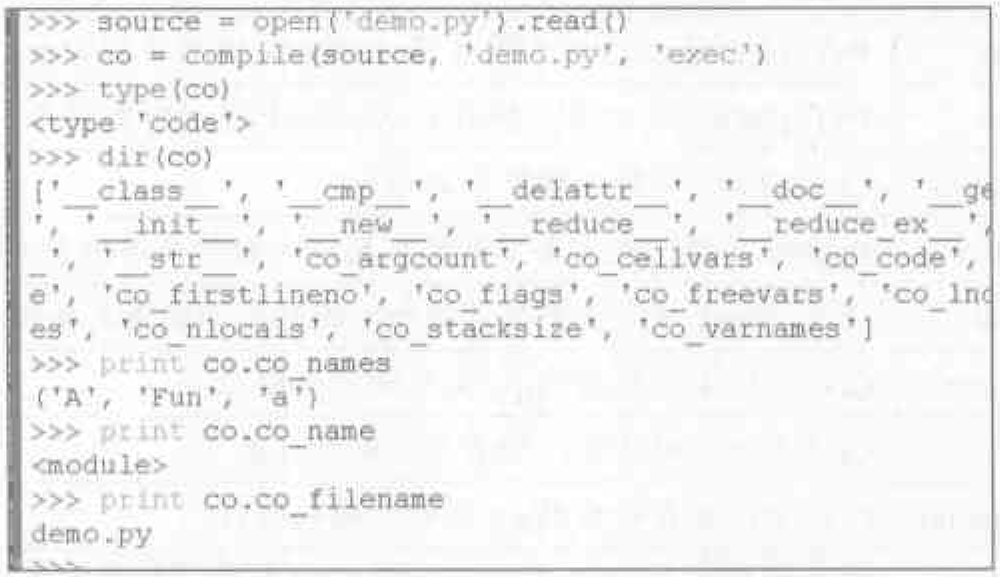
表 7-2

| 字节码在 co_code 中的偏移 | .py 文件中源代码的行号 |
| --- | --- |
| 0 | 1 |
| 6 | 2 |
| 50 | 7 |

这里有一个小小的技巧，Python 不会直接记录这些信息，但是，它会记录这些信息间的增量值，所以，对应的 `co_lnotab` 就应该是：0，1，6，1，44，5。

### 7.2.4 在 Python 中访问 PyCodeObject 对象

在 Python 中，有与 C 一级的 `PyCodeObject` 对象对应的对象——code 对象，这个对象是对 C 一级的 `PyCodeObject` 对象的一个简单包装，通过 code 对象，我们可以访问 `PyCodeObject` 对象中的各个域。图 7-3 展示了如何通过 `compile` 内建函数获得一个 code 对象，并访问其中的属性。关于内建函数 `compile` 的更多信息，请参阅 Python 文档。



```
>>> source = open('demo.py').read()
>>> co = compile(source, 'demo.py', 'exec')
>>> type(co)
<type 'code'>
>>> dir(co)
['__class__', '__cmp__', '__delattr__', '__doc__', '__ge
', '__init__', '__new__', '__reduce__', '__reduce_ex__',
_', '__str__', 'co_argcount', 'co_cellvars', 'co_code',
e', 'co_firstlineno', 'co_flags', 'co_freevars', 'co_lnc
es', 'co_nlocals', 'co_stacksize', 'co_varnames']
>>> print co.co_names
('A', 'Fun', 'a')
>>> print co.co_name
<module>
>>> print co.co_filename
demo.py
>>>
```

图 7-3　在 Python 中访问 PyCodeObject 对象中的信息

## 7.3 Pyc 文件的生成

### 7.3.1 创建 pyc 文件的具体过程

前面我们提到，Python 在通过 `import` 对 module 进行动态加载时，如果没有找到相应的 pyc 文件或 dll 文件，就会在 py 文件的基础上自动创建 pyc 文件。那么，要想了解 pyc 的格式到底是什么样的，我们只需考察 Python 在将编译得到的 `PyCodeObject` 写入到 pyc 文件中时到底进行了怎样的动作就可以了。代码清单 7-1 的函数就是我们的切入点。

**代码清单 7-1**

```
[import.c]
static void write_compiled_module(PyCodeObject *co, char *cpathname, long
  mtime)
{
    FILE *fp;
    //排他性地打开文件
    fp = open_exclusive(cpathname);
    //[1]: 写入 Python 的 magic number
    PyMarshal_WriteLongToFile(pyc_magic, fp, Py_MARSHAL_VERSION);
    //[2]: 写入时间信息
    PyMarshal_WriteLongToFile(mtime, fp, Py_MARSHAL_VERSION);
```

```
    //[3]: 写入 PyCodeObject 对象
    PyMarshal_WriteObjectToFile((PyObject *)co, fp, Py_MARSHAL_VERSION);

    fflush(fp);
    fclose(fp);
}
```

从 write_compiled_module 中可以发现，一个 pyc 文件中实际上包含了三部分独立的信息：Python 的 magic number、pyc 文件创建的时间信息，以及 PyCodeObject 对象。

首先在代码清单 7-1 的[1]处，我们看到 Python 会将 pyc_magic 这个值写入到文件的开头。实际上，pyc_magic 是 Python 所定义的一个整数值。一般来说，不同版本的 Python 实现都会定义不同的 magic number，这个值就是用来保证 Python 兼容性的。比如说要防止 Python 2.5 的运行环境加载由 Python 1.5 产生的 pyc 文件，那么只需要将 Python 2.5 和 Python 1.5 的 magic number 设为不同的值就可以了，因为 Python 在加载 pyc 文件时会首先检查这个 magic number，如果发现 Python 自身的 magic number 与待加载的 pyc 文件中记录的 magic number 不同，则会拒绝加载不兼容的 pyc 文件。

这里出现了一个问题，pyc 文件为什么会不兼容了？最主要的原因是字节码指令的变化，由于 Python 一直在不断地改进，有一些字节码指令退出了历史舞台，比如上面提到的 SET_LINENO；而另一些新的语法特性会导致加入新的字节码指令。这些都会导致 Python 的不兼容问题。

在 import.c 中，可以在源代码的注释里找到从 Python 1.5 到 Python 2.5 所有版本的 magic number，我们可以看一看 Python 2.5 所定义的 magic number：

```
[import.c]
#define MAGIC (62131 | ((long)'\r'<<16) | ((long)'\n'<<24))
static long pyc_magic = MAGIC;
```

在 pyc 文件中，紧接着 magic number 的是 pyc 文件创建的时间信息，代码清单 7-1 的[2]处完成了向 pyc 文件写入时间信息的动作。在 pyc 文件中包含时间信息可以使 Python 自动将 pyc 文件与最新的 py 文件进行同步。

假如在早上 9 点我们将 demo.py 文件编译成了 demo.pyc 文件，在下午 3 点时，我们修改了 demo.py，当 Python 执行修改后的 demo.py 时，因为存在 demo.pyc 文件，所以 Python 会首先尝试加载 demo.pyc，在加载的过程中，Python 会发现 pyc 文件的时间早于 py 文件的时间，于是就会自动重新编译 demo.py，生成新的 demo.pyc。需要指出的是，python 中写入时间的真实动作并不是如代码清单 7-1 的[2]所示，为了描述的简便，我们修改了这里的代码，有兴趣的读者可以参阅 Python 的源代码。

在代码清单 7-1 的[3]处，Python 将内存中的 PyCodeObject 对象写入了 pyc 文件，完成了 pyc 文件的创建工作。

在 write_compiled_module 中，向 pyc 文件写入数据的动作最后会集中到下面所示的几个函数中。在这里，我们假设代码只处理写入到文件，即 p->fp 是有效的情况。因此我们对代码进行了删减，删减部分，请参考 Python 源代码。

```
[marshal.c]
typedef struct {
    FILE *fp;
    int depth;
    PyObject *strings; /* dict on marshal, list on unmarshal */
} WFILE;

#define w_byte(c, p) putc((c), (p)->fp)

static void w_long(long x, WFILE *p)
{
    w_byte((char)( x      & 0xff), p);
    w_byte((char)((x>> 8) & 0xff), p);
    w_byte((char)((x>>16) & 0xff), p);
    w_byte((char)((x>>24) & 0xff), p);
}

static void w_string(char *s, int n, WFILE *p)
{
    fwrite(s, 1, n, p->fp);
}
```

在调用 PyMarshal_WriteLongToFile 时，会直接调用 w_long，w_long 会将需要写入的数据一个字节一个字节地写入到文件中。而在调用 PyMarshal_WriteObjectToFile 时，Python 会借助于另一个辅助的函数 w_object 来完成将 PyCodeObject 对象写入到 pyc 文件中的操作。要特别注意的是 PyMarshal_WriteObjectToFile 的第一个参数，这个参数正是 Python 编译出来的 PyCodeObject 对象。

w_object 的代码非常长，这里就不全部列出。实际上，w_object 的逻辑非常简单，就是对应不同的对象，比如 string、int、list、dict 等，会有不同的写的动作，然而其最终目的都是通过最基本的 w_long 或 w_string 将整个 PyCodeObject 写入到 pyc 文件中。换句话说，Python 在向 pyc 文件中写入一个 list 对象时，其实只是将 list 中所包含的数值或字符串写入了 pyc 文件中；同时这也意味着，Python 在加载 pyc 文件时，必须基于这些数值或字符串重新构造出 list 对象。

对于 PyCodeObject，很显然，w_object 会遍历 PyCodeObject 中的所有域，将这些域依次写入：

```
[marshal.c]
static void w_object(PyObject *v, WFILE *p)
{
    ……
    else if (PyCode_Check(v))
    {
```

```
        PyCodeObject *co = (PyCodeObject *)v;
        w_byte(TYPE_CODE, p);
        w_long(co->co_argcount, p);
        ……
        w_object(co->co_code, p);
        w_object(co->co_consts, p);
        w_object(co->co_names, p);
        ……
        w_object(co->co_lnotab, p);
        }
        ……
}
```

我们来想象一下，当 w_object 面对一个 PyListObject 对象时，会有什么动作？没错，如同面对 PyCodeObject 一样，w_object 还是遍历，将 PyListObject 对象中的每一个元素依次写入到 pyc 文件中：

```
[w_object() in marshal.c]
……
else if (PyList_Check(v))
    {
        //注意这里的p的类型为WFILE*
        w_byte(TYPE_LIST, p);
        n = PyList_GET_SIZE(v);
        w_long((long)n, p);
        for (i = 0; i < n; i++)
        {
            w_object(PyList_GET_ITEM(v, i), p);
        }
    }
……
```

而如果 w_object 面对的是 PyIntObject，嗯，那太简单了，几乎没有什么可说的：

```
[w_object() in marshal.c]
……
else if (PyInt_Check(v))
    {
        w_byte(TYPE_INT, p);
        w_long(x, p);
    }
……
```

细心的你一定注意到了，w_object 毫无例外地都会在写入对象之前，先写入一个 TYPE_LIST、TYPE_CODE，或者是 TYPE_INT 这样的标识。这些标志对于 pyc 文件的再次加载具有至关重要的作用。

在前面我们就已经提到过，Python 最终仅仅会将数值和字符串写入到 pyc 文件中。当 PyCodeObject 对象写入到 pyc 文件之后，所有的数据都变成了字节流，类型信息丢失了。如果没有类型信息，Python 再次加载 pyc 文件时，就再也没有办法知道这些字节流中隐藏的结构和蕴含的信息了，所以 Python 必须将对象的类型信息也写入到 pyc 文件中，这些标

识正是 Python 定义的类型信息，如果 Python 在 pyc 文件中发现这样的标识，则预示着上一个对象结束，新的对象开始，而且也知道了新对象是什么类型的对象。只有这样，当 Python 加载 pyc 文件时，加载器才能知道在什么时候应该进行什么样的加载动作。

这些标识同样也是在 import.c 中定义的：

```
[import.c]
#define TYPE_NULL    '0'
#define TYPE_NONE    'N'
……
#define TYPE_INT     'i'
#define TYPE_STRING 's'
#define TYPE_INTERNED    't'
#define TYPE_STRINGREF  'R'
#define TYPE_TUPLE   '('
#define TYPE_LIST    '['
#define TYPE_DICT    '{'
#define TYPE_CODE    'c'
```

到了这里，可以看到，Python 对于 PyCodeObject 对象的导出动作其实是不复杂的。实际上在 write 的动作中，不论面临 PyCodeObject 还是 PyListObject 这些复杂对象，最后都会归结为简单的两种形式，一个是对数值的写入，一个是对字符串的写入。上面其实已经看到了对数值的写入过程，对数值的写入非常简单，仅仅需要按字节依次写入到 pyc 文件中即可。而在写入字符串时，Python 则设计了一套比较复杂的机制。

## 7.3.2 向 pyc 文件写入字符串

在了解 Python 如何将字符串写入到 pyc 文件中的机制前，我们首先需要介绍一个在写入过程中关键的结构体 WFILE（有删节）：

```
[marshal.c]
typedef struct {
    FILE *fp;
    PyObject *strings; /* dict on marshal, list on unmarshal */
} WFILE;
```

这里我们也只考虑 fp 有效的情况，即写入到文件中。这时，WFILE 可以看作是一个对 FILE*的简单包装，但是在 WFILE 里，出现了一个奇特的 strings 域。这个域是 Python 向 pyc 文件中写入字符串或从其中读出字符串的关键所在，当向 pyc 中写入时，strings 会指向一个 PyDictObject 对象；而从 pyc 中读出时，strings 则会指向一个 PyListObject 对象。

我们先来看一看 Python 将字符串写入到 pyc 文件的过程：

```
[marshal.c]
void PyMarshal_WriteObjectToFile(PyObject *x, FILE *fp, int version)
{
```

```
    WFILE wf;
    wf.fp = fp;
    wf.strings = (version > 0) ? PyDict_New() : NULL;
    w_object(x, &wf);
}
```

可以看到，WFILE 的 strings 在真正开始将 PyCodeObject 写入到 pyc 文件之前，就已经被创建了。在 w_object 中对于字符串的处理部分，我们可以看到对 strings 的使用（见代码清单 7-2）。

代码清单 7-2

```
[w_object() in marshal.c]
......
else if (PyString_Check(v))
    {
        if (p->strings && PyString_CHECK_INTERNED(v))
        {
            //[1] : 获得 PyStringObject 对象在 strings 中的序号
            PyObject *o = PyDict_GetItem(p->strings, v);
            //[2] : intern 字符串的非首次写入
            if (o)
            {
                long w = PyInt_AsLong(o);
                w_byte(TYPE_STRINGREF, p);
                w_long(w, p);
                goto exit;
            }
            //[3] : intern 字符串的首次写入
            else
            {
                o = PyInt_FromLong(PyDict_Size(p->strings));
                PyDict_SetItem(p->strings, v, o);
                Py_DECREF(o);
                w_byte(TYPE_INTERNED, p);
            }
        }
        //[4] : 写入普通 string
        else
        {
            //写入字符串的类型 TYPE_STRING
            w_byte(TYPE_STRING, p);
        }
        //写入字符串的长度
        n = PyString_GET_SIZE(v);
        w_long((long)n, p);
        //写入字符串
        w_string(PyString_AS_STRING(v), n, p);
    }
......
```

在向 pyc 文件中写入一个字符串时，可能会发生 3 种情况。第一种情况是写入一个普通的字符串，这时的处理就非常简单了，先是写入字符串的类型标识 TYPE_STRING，然后

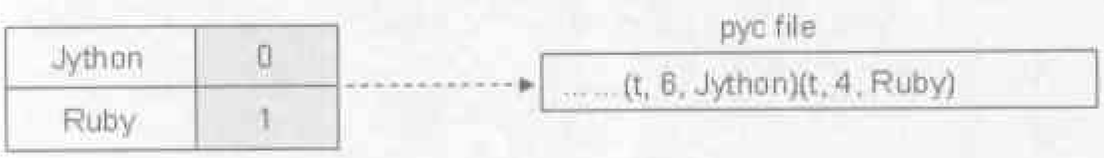
调用 w_long 写入字符串的长度，最后通过 w_string 写入字符串本身。这一切在代码清单 7-2 的[4]处完成。

在普通字符串之外，Python 还会碰到写入需要在以后加载 pyc 文件时进行 intern 操作的字符串。对这种字符串，又会分为首次写入和非首次写入的情况，这就是 Python 在写入字符串时会遇到的另外两种情况：intern 字符串的首次写入和 intern 字符串的非首次写入。

为了理解这两种情况的区别，我们需要了解 WFILE 结构中的 strings 域在 Python 向 pyc 文件写入字符串的过程中究竟扮演了一个怎样的角色。实际上，之前我们已经看到，WFILE 中的 strings 会在 PyMarshal_WriteObjectToFile 中被设置为指向一个通过 PyDict_New 创建的 PyDictObject 对象。在 strings 所指向的这个 PyDictObject 对象中，实际上维护着（PyStringObject，PyIntObject）这样的映射关系。那么这个 PyIntObject 对象的值是什么呢？这个值表示的是对应的 PyStringObject 对象是第几个被加入到 WFILE.strings 中的字符串。更准确地说，是第几个被写入到 pyc 文件中的 intern 字符串。

Python 为什么需要这个 PyIntObject 对象的值，看上去似乎有些奇怪，记录一个字符串被加入到 WFILE.strings 中的序号有什么意义呢？好，让我们来考虑下面的情形。

假设我们需要向 pyc 文件中写入 3 个 string："Jython"、"Ruby"、"Jython"，而且这 3 个 string 在以后 pyc 文件被加载时都需要进行 intern 操作。对于前两个 string，没有任何问题，按照代码清单 7-2 的[3]处的动作，闭着眼睛写入就是了。完成了前两个 string 的写入后，WFILE.strings 与 pyc 文件的情况如图 7-4 所示：




注意，pyc 文件中的括号和逗号是为了方便理解所添加。

图 7-4 写入 "Jython" 和 "Ruby" 之后的 pyc 文件

在写入第 3 个字符串的时候，麻烦来了。对于这个"Jython"，我们应该怎么处理呢？是按照上两个 string 一样吗？如果这样的话，写入后，WFILE.strings 和 pyc 的情况如图 7-5 所示：




图 7-5 强行第二次写入 "Jython" 后的 pyc 文件

我们先不管 PyDictObject 可不可能实现图 7-5 中所展示的 strings，因为不可能有相同的键出现。不看 strings，只是看一看 pyc 文件，我们就知道，问题来了。在 pyc 文件

中，出现了重复的内容，关于“Jython”的信息重复了两次，这会引起什么麻烦呢？想象一下在 Python 代码中，我们定义了符号 button，就假设一个变量名为 button，在此之后，我们又在多处使用了 button 这个变量。Python 在对 pyc 文件进行写入时，需要将变量名也写入到 pyc 文件中。这样，在 pyc 文件中，“button”将出现多次。想象一下吧，我们的 pyc 文件会变得多么臃肿，而其中充斥的只是毫无价值的冗余信息。如果你是 Python 的设计者，你能忍受这样的设计吗？当然不能。于是 Python 的设计者给了我们 TYPE_STRINGREF 这个东西。在解析 pyc 文件时，这个标志表明后面的一个数值表示了一个索引值，根据这个索引值到 WFILE.strings 中去查找，就能找到需要的 string 了，这一点，在后面我们会看得更清楚。

有了 TYPE_STRINGREF，我们的 pyc 文件就能变得苗条了，如图 7-6 所示：




图 7-6 采用了 TYPE_STRINGREF 后的 pyc 文件

好了，为了对这个过程有一个更清晰的了解，我们来总结一下。对于一个 intern 字符串，Python 会首先于代码清单 7-2 的[1]处在 strings 中查找其中是否已经记录了该字符串，这个查找动作会导致两个结果。

1. 查找失败，流程转移到代码清单 7-2 的[3]处，Python 进入 intern 字符串的首次写入，在首次写入时，Python 会进行两个独立的动作：
   - 将（字符串，序号）添加到 strings 中；
   - 将类型标识 TYPE_INTERND 和字符串本身写入到 pyc 文件中。
2. 查找成功，流程转移到代码清单 7-2 的[2]处，Python 进入 intern 字符串的非首次写入，这时，Python 仅仅只是将类型标识 TYPE_STRINGREF 和查找得到的序号写入到 pyc 文件中。

到了这里，我们有些迷惑了，好吧，看上去记录字符串插入 strings 的序号好像有些道理，但是既然 strings 是个 PyDictObject 对象，而我们知道，PyDictObject 对象是绝对没有索引访问的能力的，那么这个序号究竟还有什么用呢？

没错，这个被记录的序号，在写入 pyc 的过程中，毫无用处。真正有趣的是，这个被记录的序号是用于加载 pyc 文件的过程的。而更加有趣的是，在加载 pyc 文件时，我们同样会用到 WFILE，而这时 strings 再也不是一个 PyDictObject 对象，而是一个 PyListObject 对象了。PyListObject 是支持索引访问的，是不是有些明了了？

看一下加载 pyc 文件的过程，就能对这个机制更加地明了。前面我们提到，在读入 pyc 文件时，`WFILE.strings` 将是一个 `PyListObject` 对象，这一点在 `PyMarshal_ReadObjectFromFile` 中可以看到。

```
[marshal.c]
PyObject *PyMarshal_ReadObjectFromFile(FILE *fp)
{
    RFILE rf;
    PyObject *result;
    rf.fp = fp;
    rf.strings = PyList_New(0);
    result = r_object(&rf);
    return result;
}
```

Python 进入 `r_object` 之后，就开始从 pyc 文件中读入数据，并创建 `PyCodeObject` 对象，这个 `r_object` 是 `w_object` 的逆运算，所以，你应该可以猜想得到它的整个流程了。

当 Python 读到了 `TYPE_INTERND` 后，会将其后的字符串读入，将这个字符串进行 intern 操作，同时将 intern 操作的结果添加到 `strings` 这个 `PyListObject` 中。

随后，当 Python 从 pyc 文件中读到 `TYPE_STRINGREF` 时，会根据其后跟随的序号值访问 `strings`，从而就获得了已经进行了 intern 操作的 `PyStringObject` 对象。

所以在读入前两个字符串后，`WFILE.strings` 的情形如图 7-7 所示：




注：粗体部分表示已经加载的部分

图 7-7 加载“Jython”和“Ruby”之后的 WFILE.strings

在加载紧接着的（R，0）时，因为解析到是一个 `TYPE_STRINGREF` 标志，所以直接以标志后面的数值 0 位索引访问 `WFILE.strings`，立刻可得到字符串“`Jython`”。

## 7.3.3 一个 PyCodeObject，多个 PyCodeObject

到了这里，关于 `PyCodeObject` 与 pyc 文件，出现了一个有趣的问题。还记得前面那个 demo.py 吗？我们说那段简单到什么都做不了的 python 代码就要产生 3 个 `PyCodeObject`。而在 `write_compiled_module` 中我们又亲眼看到，Python 运行环境只会对一个 `PyCodeObject` 对象调用 `PyMarshal_WriteObjectToFile` 操作。刹那间，我们竟然看到了两个遗失的 `PyCodeObject` 对象。

Python 显然不会犯这样低级的错误，想象一下，如果你是 Python 的设计者，这个问

题该如何解决？很自然地，我们会假想，有两个 PyCodeObject 对象一定是包含在另一个 PyCodeObject 中的。没错，确实如此，还记得我们最开始指出的 Python 是如何确定一个 Code Block 的吗？对喽，就是名字空间。仔细观察一下 demo.py，你会发现在源文件中作用域就呈现出一种嵌套的结构，这种结构也正是 PyCodeObject 对象之间的结构。所以到现在清楚了，与 Fun 和 A 对应的 PyCodeObject 对象一定是包含在与 demo.py 对应的 PyCodeObject 对象中的，而 PyCodeObject 结构中的 co_consts 域正是这两个 PyCodeObject 对象的藏身之处，如图 7-8 所示：

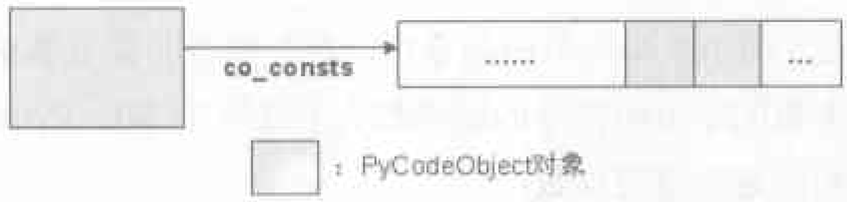



图 7-8 PyCodeObject 对象之间的嵌套结构

在对一个 PyCodeObject 对象进行写入到 pyc 文件的操作时，如果碰到它包含的另一个 PyCodeObject 对象，那么就会递归地执行写入 PyCodeObject 对象的操作。如此下去，最终所有的 PyCodeObject 对象都会被写入到 pyc 文件中去。所以，pyc 文件中的 PyCodeObject 对象也是以一种嵌套的关系联系在一起的。

这种嵌套的关系意味着 pyc 文件中的二进制数据实际是一种有结构的数据，这种结构化性质预示着我们能够以 XML 的形式来将 pyc 文件进行可视化。马上，你就可以看到这一激动人心的结果。

## 7.4 Python 的字节码

关于 Python 的编译结果，我们还剩下最后一个话题了，那就是 Python 的字节码。这里并不会对 Python 字节码进行详细的介绍，这一部分将是我们在以后的章节中对 Python 虚拟机剖析时的重点。现在我们仅仅对 Python 字节码做一个粗略的介绍。不管怎么说，现在也应该和它们打个招呼了。

我们知道，Python 源代码在执行前会被编译为 Python 的字节码指令序列，Python 虚拟机就是根据这些字节码来进行一系列的操作，从而完成对 Python 程序的执行。在 Python 2.5 中，一共定义了 104 条字节码指令：

```
[opcode.h]
#define STOP_CODE    0
#define POP_TOP      1
#define ROT_TWO      2
......
#define CALL_FUNCTION_KW          141
#define CALL_FUNCTION_VAR_KW       142
#define EXTENDED_ARG  143
```

所有这些字节码的操作含义在 Python 自带的文档中有专门的一页进行描述，当然，也可以到下面的网址察看：*http://docs.python.org/lib/bytecodes.html*。

细心的你一定发现了，虽然 Python 2.5 中只定义了 104 条字节码指令，但是字节码指令的编码却到了 143，似乎字节码指令的编码有跳跃。没错，Python 2.5 中字节码指令的编码并没有按顺序增长，比如编码为 5 的 `ROT_FOUR` 指令之后就是编码为 9 的 `NOP` 指令。这可能是历史遗留下来的，你知道，在咱们这行，历史问题可不怎么好处理，搞得现在还有许多人不得不很郁闷地面对 MFC ☺。

在 Python 2.5 的 104 条字节码指令中，有一部分是需要参数的，另一部分是没有参数的。所有需要参数的字节码指令的编码都大于或等于 90。Python 中提供了专门的宏来判断一条字节码指令是否需要参数：

```
[opcode.h]
#define HAVE_ARGUMENT 90
#define HAS_ARG(op) ((op) >= HAVE_ARGUMENT)
```

# 7.5 解析 pyc 文件

好了，到了现在，关于 `PyCodeObject` 和 pyc 文件的一切我们都已了如指掌了。前面我们提到，基于我们对 pyc 文件的了解，可以做一些非常有趣的事了。呃，说白了，就是自己写一个 pyc 文件解析器，以 XML 的形式输出解析结果，将 pyc 文件可视化。没错，利用我们现在所知道的一切，我们真的可以这么做了。在本书附带的代码中有一个名为 PycParser 的工程，在其中实现了将 pyc 文件转换为可视的 XML 文件的一个简单的方法。图 7-9 展现的是 PycParser 对本章前面的那个 demo.py 的解析结果。




图 7-9 PycParser 的解析结果

图 7-9 中显示的是 pyc 文件中的第三部分——PyCodeObject。可以看到，在 PyCodeObject 对象的 co_consts 中，包含了另外的 PyCodeObject 对象，同时还包含了别的对象，实际上在 co_consts 中，包含了 Python 源文件中所定义的所有常量对象。所谓常量对象，就是除了一些 PyStringObject 对象之外的所有对象，因为有的 PyStringObject 对象是作为符号存在的，这些符号通常保存在 co_names 和 co_varnames 中，如图 7-10 所示。

```
- <names>
    <strRef index="0" value="A" />
    <strRef index="3" value="Fun" />
    <internStr index="4" length="1" value="a" />
  </names>
  <varNames />
```

图 7-10 demo.pyc 中的符号表(co_names)和局部变量表(co_varnames)

对照 demo.py，你会发现这些对象都是其中定义的对象的名字。可以看到，其实 co_names 和 co_varnames 之间是有区别的，co_names 中会记录所有的符号，而 co_varnames 中，所记录的则是所有用于局部变量的符号。在 Python 2.4.3 中，这里的 co_varnames 是和 co_names 一样的，同样包含 A，Fun，a 这三个符号，但是在 Python 2.5 中，这一点被改变了。

将图 7-9、图 7-10 与表 7-1 中对 PyCodeObject 对象各个域的描述对照参考，就能更好地理解 PyCodeObject 对象中主要的域所包含的信息了。

到了现在，更进一步，我们还可以解析字节码指令序列。前面我们已经知道，Python 在生成 pyc 文件时，会将 PyCodeObject 对象中的字节码指令序列也写入到 pyc 文件中，而且这个 pyc 文件中还记录了每一条字节码指令与 Python 源代码行号的对应关系，嗯，就是那个 co_lnotab 啦。假如现在我们知道了字节码指令在 co_code 中的偏移地址，那么与这条字节码指令对应的 Python 源代码的位置可以通过下面的算法（Python 伪代码）得到：

```
lineno = addr = 0
for addr_incr, line_incr in c_lnotab:
    addr += addr_incr
    if addr > A:
        return lineno
 lineno += line_incr
```

下面是从一段 Python 源代码对应的 PyCodeObject 解析出字节码指令序列的结果，这个结果也将作为下一章对 Python 虚拟机的分析的开始：

```
[declare.py]
i = 1
#   LOAD_CONST    0
#   STORE_NAME    0
```

```
s = "Python"
#   LOAD_CONST    1
#   STORE_NAME    1

d = {}
#   BUILD_MAP    0
#   STORE_NAME    2

l = []
#   BUILD_LIST    0
#   STORE_NAME    3
#   LOAD_CONST    2
#   RETURN_VALUE    none
```

事实上，Python 标准库中提供了对 python 的 code 对象进行解析的工具 dis，dis 提供一个名为 dis 的方法，这个方法接收一个 code 对象，然后会输出 code 对象里的字节码指令信息。利用这个工具，可以很容易地得到我们在这里得到的结果，当然，还要更详细一些，图 7-11 展示了利用 dis 工具对 declare.py 进行解析的结果：

```
>>> s = open('declare.py').read()
>>> co = compile(s, 'declare.py', 'exec')
>>> import dis
>>> dis.dis(co)
  1           0 LOAD_CONST               0 (1)
              3 STORE_NAME               0 (i)

  2           6 LOAD_CONST               1 ('python')
              9 STORE_NAME               1 (s)

  3          12 BUILD_MAP                0
             15 STORE_NAME               2 (d)

  4          18 BUILD_LIST               0
             21 STORE_NAME               3 (l)
             24 LOAD_CONST               2 (None)
             27 RETURN_VALUE
```

图 7-11 利用 dis 工具对 declare.py 对应的 code 对象进行解析

在图 7-11 显示的结果中，最左面一列显示的是字节码指令对应的源码在 declare.py 中的行数，左起第 2 列显示的是当前的字节码指令在 co_code 中的偏移位置，第 3 列显示了当前的字节码指令，最后一列显示了当前字节码指令的参数。

在以后的分析中，我们大部分将采用 dis 工具的解析结果，在有些特殊情况下会使用我们自己的解析结果。

能够解析 pyc 文件，对于我们来说，实际上是一件非常重要的成就，想一想，从现在到达的地方出发，我们就可以做出一个 Python 的执行引擎了，嘿，这是多么激动人心的事啊。遥远的天空，一抹朝阳，缓缓升起了。

CHAPTER

第 8 章

# Python 虚拟机框架

从本章开始，我们将切入 Python 字节码虚拟机，深入剖析 Python 字节码虚拟机的运行机理。在以后提及 Python 字节码虚拟机时，我们可能会根据不同情况使用虚拟机或 Python 虚拟机等名称。

Python 的虚拟机是 Python 的核心，在.py 源代码被编译器编译为字节码指令序列之后，就将由 Python 的虚拟机接手整个工作。Python 的虚拟机会从编译得到的 `PyCodeObject` 对象中依次读入每一条字节码指令，并在当前的上下文环境中执行这条字节码指令。如此反复运行，所有由 Python 源代码所规定的动作都会如期望一样，一一展开。

## 8.1 Python 虚拟机中的执行环境

Python 的虚拟机实际上是在模拟操作系统运行可执行文件的过程，如果对可执行文件的运行机理有一个大致的了解，那么对于理解 Python 虚拟机的运行机理是相当有益的。因此，在进入 Python 的虚拟机之前，我们先来看一看在普通的 x86 的机器上，可执行文件是以一种什么方式运行的。在这里，我们主要关注运行时栈的栈帧，如图 8-1 所示：

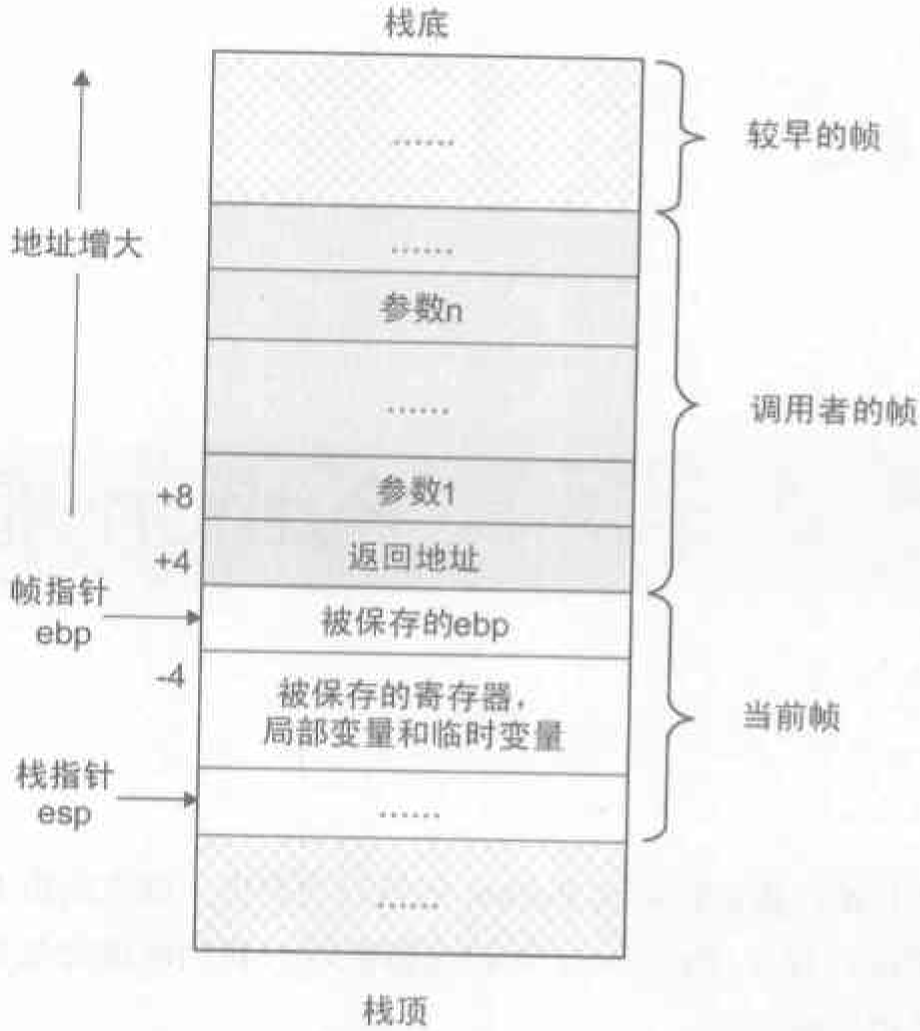



**图 8-1 可执行文件运行时的运行时栈**

图 8-1 所展示的运行时栈的情形可以看作是如下的 C 代码运行时的情形：

```
void f(int a, int b)
{
   printf("a=%d, b=%d\n", a, b);
}

void g()
{
   f(1, 2);
}

int main()
{
   g();
}
```

当程序的流程进入函数 f 时，对应于图 8-1。其中“调用者的帧”是函数 g 的栈帧，而“当前帧”则是函数 f 的栈帧。对于一个函数而言，其所有对局部变量的操作都在自己的栈帧中完成，而函数之间的调用则通过创建新的栈帧完成。

在图 8-1 所示的系统中，运行时栈是从地址空间的高地址向低地址延伸的。当在函数 g 中执行函数 f 的调用时，系统就会在地址空间中，于 g 的栈帧之后，创建 f 的栈帧。当然，在函数调用发生时，系统会保存上一个栈帧的栈指针 esp 和帧指针 ebp。当函数 f 执

行完成之后，系统会把 esp 和 ebp 的值恢复为创建 f 的栈帧之前的值，这样程序的流程又回到了函数 g 中，而程序工作的空间则又回到了函数 g 的栈帧中。这就是可执行文件在 x86 机器上的大致运行原理。而 Python 正是在其虚拟机中通过不同的实现方式模拟了这一原理，从而完成了 Python 字节码指令序列的执行。如果你对这个过程不是太清晰，不要紧，这并不妨碍你对后面内容的理解。

上一章的剖析表明 Python 源代码经过编译之后，所有的字节码指令以及程序的其他静态信息都存放在 PyCodeObject 对象中，那么 Python 的虚拟机是否就是在这个 PyCodeObject 对象上进行所有的动作呢？

答案并不唯一。PyCodeObject 对象中包含了最关键的字节码指令，以及关于程序的所有静态信息。然而有一点，是 PyCodeObject 对象没有包含，也不可能包含的。这就是关于程序运行的动态信息——执行环境。

什么是执行环境呢，考虑下面的一个例子：

```
[environment.py]
i = 'Python'

def f():
    i = 999
    print i #1

f()
print i #2
```

在上面代码中的 1 和 2 两个地方，都进行了同样的动作，即 print i。显然，它们所对应的字节码指令肯定是相同的，但是这两条语句的执行效果是不同的。这样的结果正是在执行环境的影响下产生的。在执行“1”处的 print 时，执行环境中，i 的值为 999；而在执行 2 处的 print 时，执行环境中 i 的值为“Python”。像这种同样的符号在程序运行的不同时刻对应不同的值，甚至不同类型的情况，必须在运行时动态地被捕捉和维护。这些信息是不可能在 PyCodeObject 对象中被静态地存储的。

细心的读者一定发现了，这里的执行环境和我们之前所提到的名字空间似乎是同一个东西。它们确实比较相似，但是，我们在之后将看到的执行环境的概念并不完全等同于名字空间。实际上，名字空间仅仅是执行环境的一部分，除了名字空间，在执行环境中，还包含了其他的一些信息。

结合 x86 平台运行可执行文件的机理，我们可以用这样的机理来定性地解释 environment.py 的执行过程。当 Python 开始执行 environment.py 中的第一条表达式时，Python 已经建立起了一个执行环境 A，所有的字节码指令都会在这个执行环境中执行。Python 可以从这个执行环境中获取变量的值，也可以根据字节码指令的指示修改执行环境中某个变量的值，以影响后续的字节码指令。这样的过程会一直持续下去，直到发生了函

数的调用行为。

当 Python 在执行环境 A 中执行调用函数 f 的字节码指令时，会在当前的执行环境 A 之外重新创建一个新的执行环境 B，在这个新的执行环境 B 中，有一个新的名字为“i”的对象。这个新的执行环境 B 实际上可以对应图 8-1 中所示的一个新的栈帧。

所以在 Python 真正执行的时候，它的虚拟机实际上面对的并不是一个 PyCodeObject 对象，而是另一个对象——PyFrameObject。它就是我们所说的执行环境，也是 Python 对 x86 平台上栈帧的模拟。你瞧，它的名字中还有个 Frame 呢☺。

### 8.1.1 Python 源码中的 PyFrameObject

当然，对于 Python 而言，PyFrameObject 对象不仅仅是一个我们在 x86 机器上看到的那个简简单单的栈帧，它实际上包含了其他更多的信息。请看 Python 源码中对 PyFrame-Object 的定义：

```
[frameobject.h]
typedef struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;  //执行环境链上的前一个 frame
    PyCodeObject *f_code;   //PyCodeObject 对象
    PyObject *f_builtins;   //builtin 名字空间
    PyObject *f_globals;    //global 名字空间
    PyObject *f_locals;     //local 名字空间
    PyObject **f_valuestack;    //运行时栈的栈底位置
    PyObject **f_stacktop;      //运行时栈的栈顶位置
    ……
    int f_lasti;        //上一条字节码指令在 f_code 中的偏移位置
    int f_lineno;       //当前字节码对应的源代码行
    ……
    //动态内存，维护（局部变量+cell 对象集合+free 对象集合+运行时栈）所需要的空间
    PyObject *f_localsplus[1];
} PyFrameObject;
```

从 f_back 我们可以看出一点，在 Python 实际的执行中，会产生很多 PyFrameObject 对象，而这些对象会被链接起来，形成一条执行环境链表。这正是对 x86 机器上栈帧间关系的模拟。在 x86 上，栈帧间通过 esp 指针和 ebp 指针建立了关系，使新的栈帧在结束之后能顺利回到旧的栈帧中，而 Python 正是利用 f_back 来完成这个动作。那真实的情况是不是这样呢，我们暂且按下不表。

在 f_code 中存放的是一个待执行的 PyCodeObject 对象，而接下来的 f_builtins、f_globals、f_locals 是 3 个独立的名字空间，在这里我们看到了名字空间和执行环境之间的关系。前面我们说名字空间实际上是维护着变量名和变量值之间关系的 PyDictObject

对象，所以，在这 3 个 PyDictObject 中，分别维护了 builtin 的 name、global 的 name，以及 local 的 name 与对应值之间的映射关系。在下一节，我们将给出关于名字空间的详细解析，因为它对于理解 Python 虚拟机的行为相当关键。想想前面的那段 environment.py，在执行 print i 时，首先会到 f_locals 中去寻找 PyStringObject 对象 'i'，找到了之后，将其对应的值取出，并打印出来。

在 PyFrameObject 的开头，有一个 PyObject_VAR_HEAD，这表明 PyFrameObject 是一个变长的对象，即每次创建的 PyFrameObject 对象的大小可能是不一样的。这些变动的内存是用来做什么的呢？实际上，每一个 PyFrameObject 对象都维护了一个 PyCodeObject 对象。这表明每一个 PyFrameObject 对象和 Python 源代码中的一段 Code 都是对应的，更准确地说，是和我们在研究 PyCodeObject 时提到的那个 Code Block 对应的。而在编译一段 Code Block 时，会计算出这段 Code Block 执行过程中所需要的栈空间的大小（注意，这个栈空间才是和 x86 机器上那个用于函数执行的栈空间相对应的概念）。这个栈空间的大小存储在 f_stacksize 中，而栈本身正是那段变动的内存。因为不同的 Code Block 在执行时所需的栈空间的大小是不同的，所以决定了 PyFrameObject 的开头一定有一个 PyObject_VAR_HEAD。

前面我们说 PyFrameObject 对象是对 x86 机器上单个栈帧的模拟。既然在 x86 的单个栈帧中，包含了执行计算所必需的内存空间，为什么执行计算还需要内存空间呢？举个例子：在计算 c=a+b 时，我们需要将 a 和 b 的值分别读入内存，然后计算的结果也需要存放在内存中，这些内存就是执行计算所必需的内存。当然，在 x86 上，完成一条加法操作只需要 CPU 中的寄存器即可，这里仅仅展示了在计算的过程中是需要消耗一定的内存的。所以作为对 x86 栈帧的模拟，在 PyFrameObject 中，也提供了对这些内存空间的模拟。在今后的描述中，我们将其称为运行时栈。注意，一定要将这里的“运行时栈”的概念和 x86 平台上的“运行时栈”区分开来。我们这里所谓的“运行时栈”单指运算时所需要的内存空间。

与图 8-1 所示的 x86 平台上的运行时栈对应，图 8-2 展示了 Python 虚拟机在运行时某个时刻的完整运行时环境。

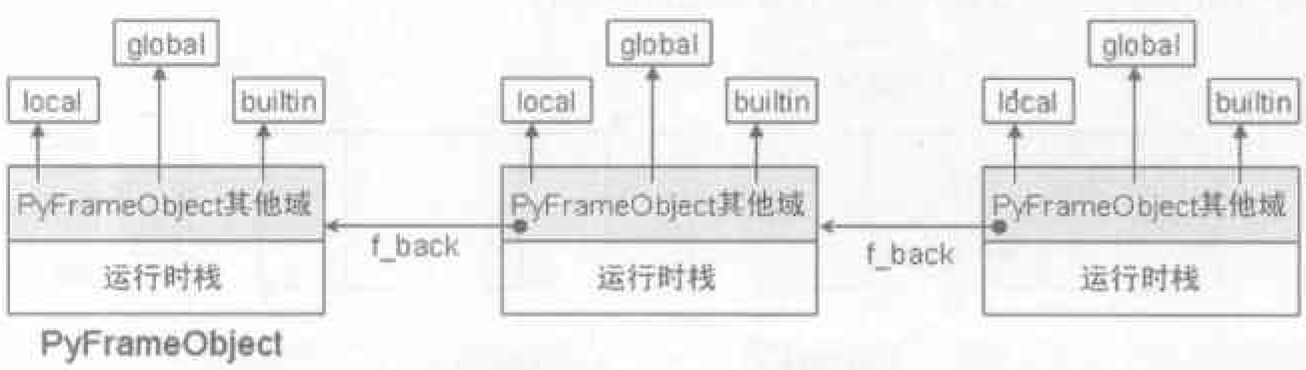



图 8-2 Python 执行的某个时刻的运行时环境

虽然在图 8-1 中连续的内存空间到图 8-2 已经变成了分离的内存空间，但是对比图 8-1 和图 8-2，我们仍然能够看出它们之间的相似之处。

## 8.1.2 PyFrameObject 中的动态内存空间

在 PyFrameObject 对象所维护的运行时栈中，存储的都是 PyObject*，可以看出，这个栈的起始位置是从 f_localsplus 开始的。其实不完全正确，f_localsplus 确实维护了一段变动长度的内存，但是这段内存不只是给栈使用的，还有别的对象也会使用：

```
[frameobject.c](有删节)
PyFrameObject *
PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject *globals,
        PyObject *locals)
{
    PyFrameObject *f;
    Py_ssize_t extras, ncells, nfrees, i;
    ncells = PyTuple_GET_SIZE(code->co_cellvars);
    nfrees = PyTuple_GET_SIZE(code->co_freevars);
    //四部分构成了 PyFrameObject 维护的动态内存区，其大小由 extras 确定
    extras = code->co_stacksize + code->co_nlocals + ncells + nfrees;
    f = PyObject_GC_NewVar(PyFrameObject, &PyFrame_Type, extras);
    //计算初始化时运行时栈的栈顶
    extras = code->co_nlocals + ncells + nfrees;
    //f_valuestack 维护运行时栈的栈底，f_stacktop 维护运行时栈的栈顶
    f->f_valuestack = f->f_localsplus + extras;
    f->f_stacktop = f->f_valuestack;
    return f;
}
```

可见，在创建 PyFrameObject 对象时，额外申请的那部分内存中有一部分是给 PyCodeObject 对象中存储的那些局部变量的、co_freevars、co_cellvars 使用的（关于 co_freevars、co_cellvars，它们涉及 Python 中对闭包的实现，在以后考察函数机制时会深入剖析），而另一部分才是给运行时栈使用的。所以，PyFrameObject 对象中的栈的起始位置（也就是栈底）是由 f_valuestack 维护的，而 f_stacktop 维护了当前的栈顶。图 8-3 是一个刚被创建的 PyFrameObject 对象的示意图。从中可以清晰地看到运行时栈和 PyFrameObject 对象中动态内存部分的关系。

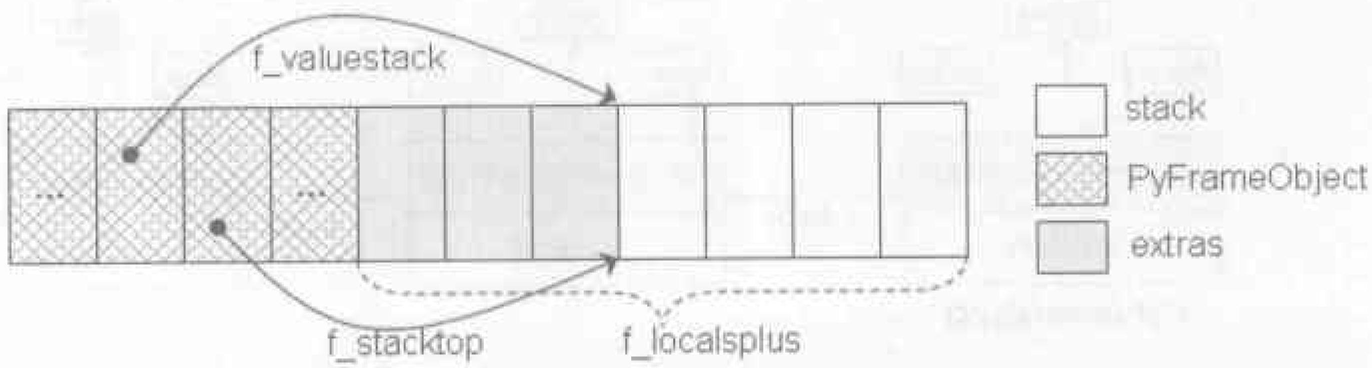



图 8-3 新创建的 PyFrameObject 对象

## 8.1.3 在 Python 中访问 PyFrameObject 对象

尽管 PyFrameObject 对象是一个用于 Python 虚拟机实现的极为隐秘的内部对象，但是 Python 还是提供了某种途径可以访问到 PyFrameObject 对象。在 Python 中，有一种 frame object，它是对 C 一级的 PyFrameObject 的包装。而且，非常幸运的是，Python 提供的一个方法能方便地获得当前处于活动状态的 frame object。这个方法就是 sys module 中的_getframe 方法。

下面的 caller.py 演示了如何利用获得当前活动的 frame object，进而获取调用当前函数的函数的信息：

```
[caller.py]
import sys

value = 3

def g():
    frame = sys._getframe()
    print 'current function is : ', frame.f_code.co_name
    caller = frame.f_back
    print 'caller function is : ', caller.f_code.co_name
    print "caller's local namespace : ", caller.f_locals
    print "caller's global namespace : ", caller.f_globals.keys()

def f():
    a = 1
    b = 2
    g()

def show():
    f()

show()
```

下面是执行 caller.py 的输出结果：

```
[caller.py 的输出结果]
F:\PythonBook\Src\frame>python caller.py
current function is :  g
caller function is :  f
caller's local namespace :  {'a': 1, 'b': 2}
caller's global namespace :  ['g', 'f', '__builtins__', '__file__', 'show',
'value', 'sys', '__name__', '__doc__']
```

从执行的结果可以看到，从函数 f 中我们完全获得了其调用者——函数 g 的一切信息，甚至包括函数 g 的各个名字空间。

有兴趣的读者可能对 sys._getframe 是如何实现的很感兴趣，下面我们就给出一个利用 Python 的异常机制实现和 sys._getframe 功能相同的代码。frame_getter.get_current_frame 的功能和 sys._getframe 的功能完全一样。

```
[frame_getter.py]
import sys
def get_current_frame():
    try:
        1/0
    except Exception, e:
        type, value, traceback = sys.exc_info()
        return traceback.tb_frame.f_back
```

# 8.2 名字、作用域和名字空间

上一节在 PyFrameObject 中，我们看到了 3 个独立的名字空间：local 名字空间、global 名字空间和 builtin 名字空间。名字空间对 Python 来说，是一个非常核心的概念，整个 Python 虚拟机运行的机制与“名字空间”这个概念有非常紧密的联系。在 Python 中，与名字空间这个概念紧密联系着的还有“名字”、“作用域”这些概念。本节将深入地介绍这些概念在 Python 中的实现或作用。

在本节中，我们将提到函数、module、class 等概念。尽管这些部分将在以后的章节中才会被剖析，但是本节将不会涉及函数、module、class 在 Python 中是如何实现的。

## 8.2.1 Python 程序的基础结构——module

现实中的 Python 程序通常并不会集中在一个巨大的.py 文件中。相反，一般来说，一个 Python 应用程序总是由多个.py 文件组成，每一个.py 文件中包含了多行 Python 中的表达式，每一个.py 文件被称 Python 视为一个 module。这些 module 中，有一个主 module，如果你的 Python 应用程序是通过 python main.py 启动的，那么这个 main.py 就是一个主 module。

Python 中引入 module 的概念，其主要目的是将一些逻辑相关的代码放到一个 module 中，以备日后使用，即实现代码复用；而另一个目的则是为整个系统划分名字空间。

一个名字（有时也称为符号）就是用于代表某些事物的一个有助于记忆的字符序列。在 Python 中，一个标识符就是一个名字，比如变量名、函数名、类名等等，这些都是名字。名字最终的作用不在于名字本身，而在于名字背后对应的那个事物。对 Python 这类动态语言来说，名字的意义远比其对 C 这样的静态语言的意义大，因为名字是 Python 在运行时能够找到其所对应的东西的唯一途径。

在 Python 中，要使用或执行一个 module，必须首先加载一个 module。加载可以用两种方式：一种是一般 module 的加载，通过 import 动作进行动态地加载；一种是主 module

的加载，通过 python main.py 这样的方式完成。不管一个 module 是如何被加载的，在加载的过程中都会进行一个动作——执行 module 中的表达式。

考虑下面的一个 module：

```
[A.py]
a = 1
a += 1
def f():
  print a
print a
```

在 module A 被加载时，Python 会执行“a = 1”、“a += 1”、“def f():”、“print a”这 4 条语句（没错，“def f()”这个貌似函数“定义”的语句确实是一个可以被执行，也必须被执行的表达式，这点与 C/C++中的函数完全不同）。在这 4 个看上去都差不多的语句中，有两个特殊的语句，它们被称为“赋值语句”。

## 8.2.2 约束与名字空间

在 Python 中，赋值语句（更确切地说，是具有赋值行为的语句）是一类相当特殊的语句，原因很简单，它们会影响名字空间。在 A.py 中，“a = 1”是一个赋值语句，它的作用是首先创建一个整数对象 1，然后将这个对象“赋给”名字 a。同样地“def f()”也是一个赋值语句，它的作用是首先创建一个函数对象（参见剖析函数机制的章节），然后将这个函数对象“赋给”名字 f。

我们可以总结出 Python 中赋值语句行为的共同之处：

- 创建一个对象 obj
- 将 obj“赋给”一个名字 name

在 Python 中，除了在 A.py 中我们见到的赋值语句外，还有如“class A(object):”，“import abc”这样的语句都是赋值语句，都遵循赋值语句的行为。

在赋值语句被执行之后，从概念上讲，我们实际上得到了一个（name，obj）这样的关联关系，对于这个关联关系，我们采用《程序设计语言——实践之路》里的术语，将之称为约束。赋值语句就是约束建立的地方。在一个约束被创建之后，它不会立刻消失，相反，它会长久地影响程序的行为。约束的容身之处就是名字空间。在 Python 中，名字空间就是一个 PyDictObject 对象实现的。约束既然是（name，obj）这样的关联关系，那么 PyDictObject 简直就是为它量身订做的。

回到我们的 A.py，在一个 module 被加载到 Python 中之后，它在内存中以一个 module 对象（参见剖析 module 实现的章节）的形式存在。在 module 对象中，维护着一个名字空

间（一个 dict 对象）。而（a, 1）、（f, function object）这些约束就位于 module 的名字空间中。

一个对象的名字空间中的所有名字都称为对象的**属性**。在前面，我们看到了 Python 中有一类“拥有赋值行为”的语句，从另一个角度来看，实际上它们也是“拥有设置对象属性的行为”的语句。既然设置了属性，那么 Python 中还有一类“拥有访问对象属性的行为”的语句，我们将访问对象属性这个动作称之为“**属性引用**”。比如对于 A.py，如果另一个 B.py 中，有“import A”和“print A.a”两条语句，其中的“A.a”就是一个属性引用。属性引用就是使用另一个名字空间中的名字，一个 module 定义了一个独立的名字空间，在另一个 module 中，要使用别的 module 中的名字，只能通过属性引用的方式访问别的 module 的名字空间，获得名字对应的对象（注意，对于 Python 中的 class 和 class 对应的实例对象，也有类似的考量）。

在 Python 中，module 之间的名字空间规则是很清晰的，但在 module 内部，对名字空间的使用有着另一套不同的规则。

## 8.2.3 作用域与名字空间

在 8.2.2 节中，我们提到，约束一旦被创建，就会被放入名字空间中，然后影响程序的行为。在 module 内部，这样的描述是没错的，但是还不够细致。在 module 内部，名字空间存在着一个可见性的问题。我们来考虑下面的例子（见代码清单 8-1）。

**代码清单 8-1**

```
[B.py]
1: a = 1 //[1]
2: def f():
3:     a = 2 //[2]
4:     print a //[3]: 输出结果为 2
5: print a //[4]: 输出结果为 1
```

在代码清单 8-1 的[3]处的输出结果为 2，[4]处的输出结果为 1，这个结果意味着在代码清单 8-1 的[1]和[2]处的两个赋值语句是在不同的名字空间中创建了约束，而[3]和[4]处的“print a”也使用了不同的名字空间中的名字“a”。

在一个 module 内部，可能存在多个名字空间，每一个名字空间都与一个作用域对应。一个约束起作用的那一段程序正文区域称为这个**约束的作用域**。而一个**作用域**则是指一段程序正文区域，在这个区域里，可能有很多个约束在起作用，一旦出了这个正文区域，这些约束都不起作用了。在 B.py 中，第 3 行和第 4 行程序正文就组成了一个作用域，在这个作用域中，“a = 2”这个约束起作用，从而影响代码清单 8-1 的[3]处的输出：“a = 2”

这个约束不能影响[4]的输出，因为第 5 行代码不在函数 `f` 所定义的作用域之内，所以“`a = 2`”这个约束不起作用。

对于作用域这个概念，至关重要的是要记住它仅仅是由源程序的文本决定的。在 Python 中，一个约束在程序正文的某个位置是否起作用，是由该约束在文本中的位置是否唯一决定的，而不是在运行时动态决定的。因此，Python 是具有**静态作用域**（也称**词法作用域**）的。而名字空间就是与作用域对应的动态的东西，一个由程序文本定义的作用域在 Python 程序运行时就会转化为一个名字空间，一个内存中的 `PyDictObject` 对象。也就是说，在函数 `f` 执行时，Python 会为 `f` 创建一个名字空间，这一点在以后剖析函数机制时会详细介绍。

位于一个作用域中的代码可以直接访问作用域中出现的名字，所谓“**直接访问**”，就是指不用加上属性引用方式的访问修饰符“.”：比如在 B.py 中访问 A.py 中的名字 `a`，需要使用“`print A.a`”的方式；而在 B.py 中访问自己这个 module 中的名字 `a`，则直接使用“`print a`”这样的方式就可以了。

访问名字这样的行为被称为**名字引用**，名字引用的规则决定了 Python 程序的行为。还是考虑 B.py，如果我们删除第 3 行代码，那么第 4 行代码的输出结果会如何呢？我们已经知道，赋值语句实际上创建了约束，删除第 3 行代码，意味着在函数 `f` 定义的作用域中，没有了与名字 `a` 相对应的约束。换句话说，在调用函数 `f` 时，名字空间中没有名字 `a` 了。那么“`print a`”的行为该如何定义呢？

一种方案是抛出异常，显然，这是非常糟糕的方案；而另一种方案是使用当前作用域（函数 `f` 定义的作用域）之外的名字 `a`，那么输出的结果就该为 1。Python 选择了第二种方案，也就是说，Python 支持嵌套作用域。

前面我们提到，module 本身关联着一个名字空间，所以 module 对应的程序正文，即 B.py 自身就是一个作用域，而 B.py 中的函数 `f` 定义的作用域位于 B.py 定义的作用域之内，这样的情况，就称为**嵌套作用域**。Python 的名字引用的行为被它所支持的嵌套作用域影响，产生的就是**最内嵌套作用域规则**：由一个赋值语句引进的名字在这个**赋值语句**所在的作用域里是可见（起作用）的，而且在其内部嵌套的每个作用域里也可见，除非它被嵌套于内部的，引进同样名字的另一条赋值语句所**遮蔽**。

为了找到某个给定名字所引用的对象，应该用这个名字在当前的作用域（名字空间）里查找。如果在这里找到了对应的约束，它就是与这个名字相关的活动约束。否则，就应该到直接的外围作用域（名字空间）去查找，并继续向外顺序地检查外围作用域（名字空间），直到到达程序的最外嵌套层次。这个最外嵌套层次就是 module 自身所定义的那个作用域。

### 8.2.3.1 LGB 规则

最内嵌套作用域规则看上去很枯燥，所以我们从这一节开始，陆续给出一些例子，形象地看一看作用域规则如何影响 Python 的行为。在 Python 中，一个 module 对应的源文件定义了一个作用域，这个称为 global 作用域（对应 global 名字空间）；一个函数定义了一个 local 作用域（对应于 local 名字空间）；Python 自身还定义了一个最顶层的作用域——builtin 作用域（对应于 builtin 名字空间，在这里定义了 Python 的 `builtin` 函数，比如 `dir`、`open`、`range` 等）。这 3 个作用域在 Python 2.2 之前就已经存在，所以那时 Python 的作用域规则被称为 LGB 规则：名字引用动作沿着 local 作用域、global 作用域、builtin 作用域的顺序查找名字对应的约束。

还是考虑 B.py 中的[3]（见代码清单 8-1），这里有一个对名字 `a` 的引用动作，因此，Python 首先会在函数 `f` 定义的作用域中（local 作用域）中查找名字 `a`，如果找到当然是最好的。如果找不到， Python 就会在 B.py 定义的作用域（global 作用域）中查找名字 `a`，如果还是找不到，则会到 Python 自身定义的 builtin 作用域中查找。图 8-4 显示了这个名字引用的过程：

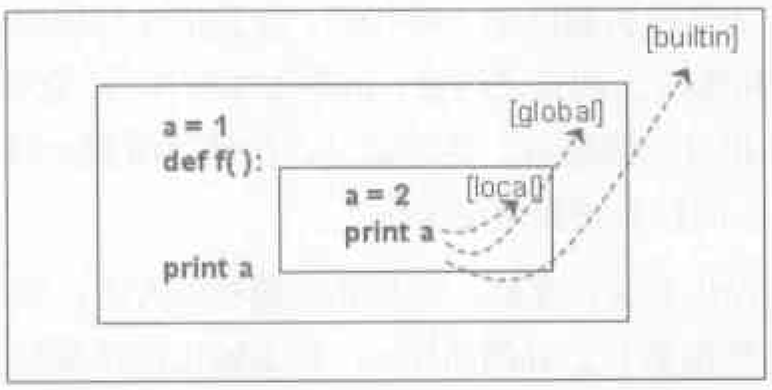



图 8-4 Python2.2 之前的 LGB 作用域规则

LGB 有一些变化的情况，比如对于 B.py 中的[4]（见代码清单 8-1），其实也会遵循 LGB 规则，只不过这时的 local 作用域和 global 作用域就是同一个作用域了。对应到名字空间上，就是同一个名字空间。更进一步，对应到 `PyFrameObject` 中，`f_local` 和 `f_global` 就是指向同一个 `PyDictObject` 对象了。

### 8.2.3.2 LEGB 规则

从 Python 2.2 开始，Python 引入了嵌套函数，这时的作用域规则才更接近我们前面提到的最内嵌套作用域规则，如代码清单 8-2 所示。

代码清单 8-2

```
[C.py]
a = 1
def f():
    a = 2
```

```
    def g():
        print a //[1]: 输出结果为 2
    return g

func = f()
func()//[2]
```

代码清单 8-2 的[2]处调用的函数实际上调用的是函数 f 中定义的内嵌函数 g，在代码清单 8-2 的[1]处，函数 g 内的“print a”的输出结果为 2。初看上去有些疑问，因为函数 f 内的约束“a = 2”在其之外应该是不起作用的，当执行 func()时，起作用的约束应该是“a = 1”才对。但是我们之前说到了，作用域仅仅是由文本决定的，函数 g 位于函数 f 之内，所以函数 g 定义的作用域内嵌于函数 f 的作用域之内。换句话说，函数 f 的作用域是函数 g 的作用域的直接外围作用域，所以，按照最内嵌套作用域规则，[1]处的名字引用应该引用的是函数 f 定义的作用域中所创建的约束。

尽管在代码清单 8-2 的[2]处，“a = 2”这个约束已经不起作用了，但是 Python 在执行“func = f()”时，会执行函数 f 中的“def g():”语句，这时 Python 会将约束“a = 2”与函数 g 对应的函数对象捆绑在一起，将捆绑后的结果返回，这个捆绑起来的整体被称为“闭包”。

实际上这里有一个相当微妙的问题，最内嵌套作用域规则是“闭包”的结果呢，还是“闭包”是最内嵌套作用域规则的实现方案？这两个问题看上去是一致的，但却隐含着谁决定谁的关系。实际上，Python 实现闭包是为了实现最内嵌套作用域规则。换句话说，最内嵌套作用域规则是语言设计时的设计策略，即是形而上的“道”；而闭包则实现语言时的一种方案，即是形而下的“器”。

这里显示的作用域规则通常也被称为 LEGB 规则，其中的 E 为 enclosing 的缩写，代表的正是“直接外围作用域”这个概念。

#### 8.2.3.3 global 表达式

初学 Python 时，由于大家不清楚 Python 的作用域规则，所以会对一些出错的情况百思不得其解。比如下面的例子：

```
a = 1

def g():
  print a

def f():
  print a //[1]
  a = 2 //[2]
  print a

g()
f()
```

运行的结果会抛出异常，显示代码清单 8-2 的[1]处有错，异常的信息为“local variable 'a' referenced before assignment”。什么？a 没有赋值？不对呀，在 module 定义的作用域内明明已经建立了“a = 1”这个约束，按照 LEGB 规则，这个输出结果应该是 1 才对，而且在调用 f 之前我们调用了函数 g，g 可是已经老老实实地输出了结果，怎么到了函数 f 里，同样的代码，就说没有赋值了呢？

理解这个错误的关键在于深刻理解最内嵌套作用域规则，这个规则的第一句话就是这个奇怪问题的症结所在。“由一个赋值语句引进的名字在这个赋值语句所在的作用域里是可见（起作用）的”，这句话的意思对应到这里，就是说虽然“a = 2”这个约束是在“print a”之后建立的，但是由于它们在同一个作用域内，所以在代码清单 8-2 的[1]处，“a = 2”这个约束的名字 a 就是可见的，按照 LEGB 规则，在 local 名字空间中就能找到名字 a，所以使用的是 local 名字空间中的 a 所对应的对象。但是很不幸的是，虽然名字 a 在[1]处已经可见了，但是要到[2]处对这个名字的赋值动作才会发生，a 才会引用一个有效的对象，所以在[1]处当然应该抛出一个“referenced before assignment”的异常。

更为有趣的东西隐藏在编译之后的字节码中，我们可以看看上面的代码反汇编的结果，如下所示：

```
a = 1

def g():
  print a
  #0  LOAD_GLOBAL       0   (a)
  #3  PRINT_ITEM
  #4  PRINT_NEWLINE

def f():
  print a
  #0  LOAD_FAST         0   (a)
  #3  PRINT_ITEM
  #4  PRINT_NEWLINE
  a = 2
```

对于相同的“print a”，Python 竟然编译出了不同的字节码指令，在函数 g 中，名字引用对应的字节码指令是 LOAD_GLOBAL，意思是要在 global 名字空间中查找名字；而在函数 f 中，名字引用对应的字节码指令为 LOAD_FAST，这条指令是指在 local 名字空间中查找名字，也就是说 Python 在编译时就已经知道名字究竟藏身于何处。这正说明了 Python 采用的是静态作用域规则，仅仅根据程序正文就能确定名字引用策略。同时，这个现象又一次地说明了最内嵌套作用域规则是指导 Python 实现（这一次是编译器的实现）的“道”。

上面的例子表明，一旦作用域中有了对于某个名字的赋值操作，这个名字就会在作用域中可见，就会出现在 local 名字空间中。换句话说，就遮蔽了外围作用域的相同的名字。

但是有的时候，我们就是想在函数 f 中输出外围作用域的名字 a，同时还要对 a 进行赋值，但是这个赋值操作在我们的设想中应该改变外围作用域中的名字 a 对应的对象，Python 精心地为我们准备了 global 关键字。当一个作用域中出现了 global 语句时，就意味着我们强制命令 Python 对某个名字的引用只参考 global 名字空间，而不用再去管 LEGB 规则。看了下面两个例子，你就会对 glboal 语句了如指掌了：

```
a = 1

def f():
  global a
  print a //输出结果: 1
  a = 2

f()
print a //输出结果: 2
```

```
a = 1

def f():
  a = 2
  def g():
      global a
      print a //输出结果: 1
      a += 1
  return g

g = f()
g()
print a //输出结果: 2
```

#### 8.2.3.4 属性引用与名字引用

属性引用实质上也是一种名字引用，其本质都是到名字空间中去查找一个名字所引用的对象。但是属性引用可以视为一种特殊的名字引用，它不受 LEGB 规则的制约。这不是说它功能更强大了，而是说它的功能更弱了。在属性引用时，一定会有对象存在，而属性引用就是到对象的名字空间中去查找名字，这里没有嵌套的作用域，它所要遵循的规则比名字引用所要遵循的 LEGB 规则简单多了。属性引用肚里可没那么多花花肠子，还要到什么外围去查找，不用了，有就是有，没有就是没有，简单明了。通常在一个 Python 程序中会同时存在属性引用和名字引用，下面的例子标出了这些不同的引用：

```
import sys

msg = 'hello world'

class A(object):
  def set(self, name):
```

```
        self.name = name  //1.对 self 中 name 的属性引用 2.对 name 的名字引用

    def show(self, show_name):
        if show_name: //对 show_name 的名字引用
            print self.name //对 self 中 name 的属性引用
        else:
            print msg //对 msg 的名字引用

a = A() //对 A 的名字引用
a.set('python')//1. 对 a 的名字引用; 2. 对 a 中 set 的属性引用
a.show(False)  //1. 对 a 的名字引用; 2. 对 a 中 show 的属性引用
```

在 8.2.1 节中提到，module 为 Python 应用程序划分了名字空间，通过属性引用，我们就可以访问各个独立名字空间中的名字；而通过名字引用，我们可以访问本 module 内部定义的多个嵌套的名字空间（作用域）。这两种方式的结合使我们能访问任何一个名字空间，但是它们的结合也给初识 Python 的人带来相当的惊奇体验。看代码清单 8-3 的例子。

**代码清单 8-3**

```
[module1.py]
import module2

owner = 'module1'
module2.show_owner() //[1]

[module2.py]
owner = 'module2'

def show_owner():
  print owner //[2]
```

在代码清单 8-3 的[1]处，发生了两次引用：首先，Python 通过名字引用获得了名字 `module2` 对应的 module 对象；然后，Python 通过属性引用获得了 `module2` 对应的 module 对象中的名字 `show_owner` 对应的函数对象。在调用函数的过程中，Python 的执行流程到达代码清单 8-3 的[2]时，发生了一次名字引用，寻找名字 `owner`，由于名字引用是不能访问自身 mudule 之外的名字空间，所以按照 LEGB 规则，[2]处输出的结果是`'module2'`。尽管 `module1` 中在调用 `show_owner` 函数之前，在 `module1` 的名字空间中引入了名字 `owner`，但是这对 `module2` 中的函数一点影响都没有，因为函数是在 `module2` 中，而名字引用遵循的 LEGB 的规则不会越过 module 的边界。

那么有没有办法让 `show_owner` 能输出 `module1` 中的 `owner` 信息呢？有的，通过函数参数，我们可以修改代码，如代码清单 8-4 所示。

**代码清单 8-4**

```
[module1.py]
import module2
```

```
owner = 'module1'
module2.show_owner(owner) //[1]

[module2.py]
owner = 'module2'

def show_owner(owner):
  print owner //[2]
```

这次在代码清单 8-4 的[2]处输出的结果就是“module1”了。有趣的是，[2]处还是一个名字引用，还是遵循着 LEGB 规则。关键之处在于，函数的参数的传递机制是一种“拥有赋值行为”的动作。什么意思呢？也就是说，函数的参数也创建了一个约束，参数名将作为名字出现在函数的 local 名字空间中（机理上可以这么理解，实际实现并非如此，参考函数机制一章），而其对应的对象则是从 module1 中传递过来的 module1. owner 对应的字符串对象。从而使 LEGB 规则能够正确地引用到字符串对象“module1”。

这又是最内嵌套作用域规则的杰作，经过对那么多实例的研究探讨，我们可以发现，这条规则对于理解 Python 的行为至关重要。正是这条规则，决定了 Python 行为的更多是代码出现的位置，而非代码执行的时间。

作用域和名字空间概念和规则对于理解 Python 的运行时行为非常关键，但是，之前我们就已经提到，它们只是用于指导 Python 如何实现的“道”，Python 的源码中又是如何实现这些形而上的“道”的呢？这些都会在随后的章节中依次展开讨论。对于现在的我们来说，谈论这些是如何实现的还为时尚早，毕竟，我们现在还不知道 Python 虚拟机是如何执行源文件编译后的字节码指令序列的。

## 8.3 Python 虚拟机的运行框架

当 Python 启动后，首先会进行 Python 运行时环境的初始化。注意这里的**运行时环境**是一个与上一节剖析的**执行环境**不同的概念。运行时环境是一个全局的概念，而执行环境实际就是一个栈帧，是一个与某个 Code Block 对应的概念。这里不明白两者的区别不要紧，在以后剖析运行时环境初始化时我们就能弄清楚两者的区别和联系。运行时环境的初始化过程非常地复杂，后面将用单独的一章来剖析，这里假设初始化的动作已经完成，我们已经站在了 Python 虚拟机的门槛外，只需要轻轻推动一下第一张骨牌，整个执行过程就像多米诺骨牌一样，一环扣一环地展开。

这个推动第一张骨牌的地方在一个名叫 PyEval_EvalFrameEx 的函数中，这个函数实际上就是 Python 的虚拟机的具体实现，它是一个非常巨大的函数，因此我们在列出其中的源代码时和以前有些不同。

PyEval_EvalFrameEx 首先会初始化一些变量，其中 PyFrameObject 对象中的 PyCodeObject 对象包含的重要信息都被照顾到了。当然，另一个重要的动作就是初始化了堆栈的栈顶指针，使其指向 f->f_stacktop：

```
[PyEval_EvalFrameEx in ceval.c]
    co = f->f_code;
    names = co->co_names;
    consts = co->co_consts;
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + co->co_nlocals;
    first_instr = (unsigned char*)PyString_AS_STRING(co->co_code);
    next_instr = first_instr + f->f_lasti + 1;
    stack_pointer = f->f_stacktop;
    f->f_stacktop = NULL;   /* remains NULL unless yield suspends frame */
```

前面我们说过，在 PyCodeObject 对象的 co_code 域中保存着字节码指令和字节码指令的参数，Python 虚拟机执行字节码指令序列的过程就是从头到尾遍历整个 co_code、依次执行字节码指令的过程。在 Python 的虚拟机中，利用 3 个变量来完成整个遍历过程。co_code 实际上是一个 PyStringObject 对象，而其中的字符数组才是真正有意义的东西，这也就是说，整个字节码指令序列实际上就是一个在 C 中普普通通的字符数组。因此，遍历过程中所使用的这 3 个变量都是 char*类型的变量：first_instr 永远指向字节码指令序列的开始位置；next_instr 永远指向下一条待执行的字节码指令的位置；f_lasti 指向上一条已经执行过的字节码指令的位置。图 8-5 展示了这 3 个变量在遍历中某时刻的情形：

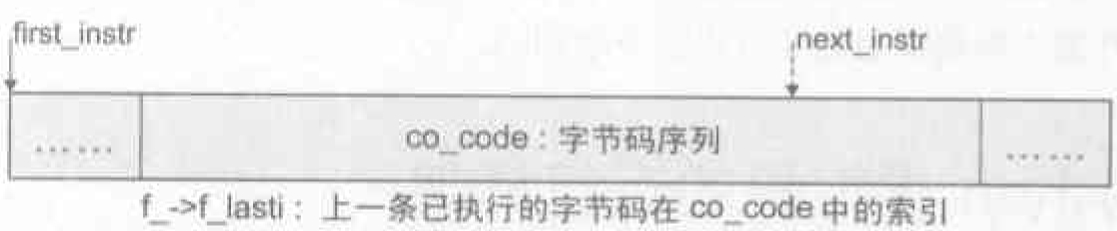



图 8-5 遍历字节码指令序列

那么这个一步一步的动作是如何完成的呢，我们来看一看 Python 虚拟机执行字节码指令的整体架构，其实就是一个 for 循环加上一个巨大的 switch/case 结构，熟悉 Windows SDK 编程的朋友可以想象一下 Windows 下那个巨大的消息循环，就是那样的结构：

```
[ceval.c]
/* Interpreter main loop */
PyObject* PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ......
    why = WHY_NOT;
    ......
    for (;;) {
    ......
    fast_next_opcode:
        f->f_lasti = INSTR_OFFSET();
        //获得字节码指令
```

```
        opcode = NEXTOP();
        oparg = 0;
        //如果指令需要参数，获得指令参数
        if (HAS_ARG(opcode))
            oparg = NEXTARG();
    dispatch_opcode:
        switch (opcode) {
        case NOP:
            goto fast_next_opcode;
        case LOAD_FAST:
            ……
        }
}
```

注意，这只是一个极度简化之后的 Python 虚拟机的样子，如果想一睹 Python 虚拟机的尊容，请参考 ceval.c 中的源码。

在这个执行架构中，对字节码的一步一步地遍历是通过几个宏来实现的：

```
[PyEval_EvalFrameEx in ceval.c]
#define INSTR_OFFSET()  (int(next_instr - first_instr))
#define NEXTOP()    (*next_instr++)
#define NEXTARG()   (next_instr += 2, (next_instr[-1]<<8) + next_instr[-2])
```

在对 PyCodeObject 对象的分析中我们说过，Python 的字节码有的是带参数的，有的是没有参数的，而判断是否带参字节码是通过 HAS_ARG 这个宏实现的。注意，对不同的字节码指令，由于存在是否需要指令参数的区别，所以 next_instr 的位移可能是不同的。但是无论如何，next_instr 总是指向 Python 下一条要执行的字节码，这很像 x86 平台上的那个 PC 寄存器。

Python 在获得了一条字节码指令和其需要的指令参数后，会对字节码指令利用 switch 进行判断，根据判断的结果选择不同的 case 语句，每一条字节码指令都会对应一个 case 语句。在 case 语句中，就是 Python 对字节码指令的实现。

在成功执行完一条字节码指令后，Python 的执行流程会跳转到 fast_next_opcode 处，或者是 for 循环处，不管如何，Python 接下来的动作都是获得下一条字节码指令和指令参数，完成对下一条指令的执行。如此一条一条地遍历 co_code 中包含的所有字节码指令，最终完成了对 Python 程序的执行。

需要提到的一点是那个名叫"why"的神秘变量，它指示了在退出这个巨大的 for 循环时 Python 执行引擎的状态。因为 Python 执行引擎不一定每次执行都会正确无误，很有可能在执行到某条字节码的时候，产生了错误，这就是我们熟悉的那个"异常"——exception。所以在 Python 退出了执行引擎的时候，就需要知道执行引擎到底是因为什么原因结束了对字节码指令的执行。是正常结束呢？还是因为有错误发生，实在是执行不下去了？why 义无反顾地担负起这一重任。关于 why 在 Python 虚拟机中作用的详细剖析，我们留到剖

析异常机制时详细讲述。

变量 `why` 的取值范围在 ceval.c 中被定义，其实也就是 Python 结束字节码执行时的状态：

```
[ceval.c]
/* Status code for main loop (reason for stack unwind) */
enum why_code {
        WHY_NOT =   0x0001, /* No error */
        WHY_EXCEPTION = 0x0002, /* Exception occurred */
        WHY_RERAISE =   0x0004, /* Exception re-raised by 'finally' */
        WHY_RETURN =    0x0008, /* 'return' statement */
        WHY_BREAK = 0x0010, /* 'break' statement */
        WHY_CONTINUE =  0x0020, /* 'continue' statement */
        WHY_YIELD = 0x0040  /* 'yield' operator */
};
```

现在，想必大家已经对 Python 的执行引擎的大体框架了然于胸了。在 Python 的执行流程进入了 `PyEval_EvalFrameEx` 中的那个 `for` 循环，取出第一条字节码之后，第一张多米诺骨牌已经被推倒，命运不可阻挡地降临了。一条接一条的字节码像潮水一样汹涌而来，浩浩荡荡，横无际涯。

# 8.4 Python 运行时环境初探

到这里，我们已经看到了 Python 虚拟机的整体的执行框架，我们还看到了 Python 虚拟机在执行时需要不断使用的执行环境。了解这两点对掌握第二部分的内容已经足够了。但是，虚拟机和执行环境还仅仅是 Python 运行机理（或者说运行模型）的一部分，为了对 Python 整个的运行机理做一个全面的了解，我们还需要大致了解一下 Python 的运行时环境。

前面我们说了，`PyFrameObject` 对应于可执行文件在执行时的栈帧，但是一个可执行文件要在操作系统中运行，只有栈帧是不够的。之前我们遗漏了两个对于可执行文件运行至关重要的概念：进程和线程。

在本节中，我们首先要对 Python 的运行模型（主要是线程模型）进行一个整体概念上的了解，虽然这部分内容我们会留到剖析 Python 的多线程实现时再详细考察，但是由于 Python 在初始化时会创建一个主线程，所以其运行时环境中存在一个主线程，而且本部分将剖析的 Python 的异常机制会利用到 Python 内部的线程模型，因此对 Python 线程模型有一个整体概念上的了解也是必须的。

以 Win32 平台为例，我们知道，对于原生的 Win32 可执行文件，无论是由 C/C++产生，还是由 Delphi 产生，都会在一个进程（Process）中运行。进程并非是与机器指令序列

相对应的活动对象，这个与可执行文件中机器指令序列对应的活动对象是由线程（Thread）这个概念来进行抽象的，而进程则是线程的活动环境。

对于通常的单线程可执行文件，在执行时操作系统会创建一个进程，在进程中，又会有一个主线程；而对于多线程的可执行文件，在执行时会操作系统会创建一个进程和多个线程。该多个线程能共享进程地址空间中的全局变量，这就自然而然地引出了线程同步的问题。CPU 对任务的切换实际上是在线程之间切换，在切换任务时，CPU 需要执行线程环境的保存工作，而在切换至新的线程之后，需要恢复该线程的线程环境。

这些关于程序运行的概念同样适用于 Python，Python 实现了对多线程的支持，而且 Python 中的一个线程就是操作系统上的一个原生线程。这里我们对多线程机制不过多深入，现在只需记住，Python 在执行时，可能会有多个线程存在。

在前面我们看到了虚拟机的大致运行框架，实际上这个虚拟机就是 Python 中对 CPU 的抽象，可以看做是一个软 CPU，Python 中的所有线程都使用这个软 CPU 来完成计算工作。真实机器上的任务切换机制对应到 Python 中，就是使不同的线程轮流使用虚拟机的机制。

CPU 切换任务时需要保存线程运行环境。对于 Python 来说，在切换线程之前，同样需要保存关于当前线程的信息。在 Python 中，这个关于线程状态信息的抽象是通过 `PyThreadState` 对象来实现的，一个线程将拥有一个 `PyThreadState` 对象。所以从另一种意义来说，这个 `PyThreadState` 对象也可以看成是对线程本身的抽象。但实际上，这两者是有很大区别的，`PyThreadState` 并非是对线程本身的模拟，因为 Python 中的线程仍然使用操作系统的原生线程。`PyThreadState` 仅仅是对线程状态的抽象，不过在本书的大部分章节中，为了叙述的方便，我们不过分严格地区分线程和线程状态本身，所以在以后我们有时会称 `PyThreadState` 为线程对象，有时会称之为线程状态对象。只有在剖析多线程机制时，我们会严格区分两者。对于下面将提到的 `PyInterpreterState` 对象，也有类似的考量。

刚才提到，在 Win32 下，线程是不能独立存活的，它需要存活在进程的环境中，而多个线程可以共享进程的一些资源。在 Python 中同样也是如此，考虑一下，如果 Python 程序中有两个线程，都会进行同样的一个动作——`import sys`，那么这个 sys module 究竟应该存在几份？是全局共享的还是每个线程都有一个 sys module？如果每个线程有自己独立 module 集合，那么 Python 对内存的消耗就会显得非常惊人。所以在 Python 中，这些 module 都是全局共享的，仿佛这些 module 都是进程中的共享资源一样，对于进程这个抽象概念，Python 以 `PyInterpreterState` 对象来实现。

在 Win32 下，通常都会有多个进程，而 Python 实际上也可以有多个逻辑上的 interpreter

存在。在通常的情况下，Python中只有一个interpreter，这个interpreter中维护了一个或多个PyThreadState对象，与这些PyThreadState对象对应的线程轮流使用一个字节码执行引擎。看，是不是与真实机器上的程序执行模型非常相似？

谈到多线程，就不能不谈到线程同步。在Python中，是通过一个全局解释器锁GIL（Global Interpreter Lock）来实现线程同步的，关于这部分内容，我们留到剖析Python多线程机制时再详细考察。

好了，现在讨论刚才提到的那两个关键对象：表示进程概念的PyInterpreterObject对象和表示线程概念的PyThreadState对象。

```
[pystate.h]
typedef struct _is {
    struct _is *next;
    struct _ts *tstate_head; //模拟进程环境中的线程集合

    PyObject *modules;
    PyObject *sysdict;
    PyObject *builtins;
    ......
} PyInterpreterState;

typedef struct _ts {
    struct _ts *next;
    PyInterpreterState *interp;
    struct _frame *frame; //模拟线程中的函数调用堆栈
    int recursion_depth;
    ......
    PyObject *dict;
    ......
    long thread_id;
} PyThreadState;
```

在PyThreadState对象中，我们看到了熟悉的PyFrameObject(_frame)对象。也就是说，在每个PyThreadState对象中，会维护一个栈帧的列表，以与PyThreadState对象的线程中的函数调用机制对应。在Win32上，情形也是一样的，每个线程都会有一个函数调用堆栈。

关于PyInterpreterState和PyThreadState的创建留待以后在合适的地方描述，不过这里可以看看PyThreadState和PyFrameObject之间的一些交互和联系。当Python虚拟机开始执行时，会将当前线程状态对象中的frame设置为当前的执行环境（frame）：

```
[ceval.c]
PyObject *PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ......
    //通过 PyThreadState_GET 获得当前活动线程对应的线程状态对象
    PyThreadState *tstate = PyThreadState_GET();
    ......
```

```
    //设置线程状态对象中的 frame
    tstate->frame = f;
    co = f->f_code;
    names = co->co_names;
    consts = co->co_consts;
    ......
    //虚拟机主循环
    for (;;) {
        ......
        opcode = NEXTOP();
        oparg = 0;
        if (HAS_ARG(opcode))
        oparg = NEXTARG();
dispatch_opcode:
        //指令分派
        switch (opcode) {
                ......
          }
        ......
    }
    ......
}
```

而在建立新的 PyFrameObject 对象时，则从当前线程的状态对象中取出旧的 frame，建立 PyFrameObject 链表：

```
[frameobject.c]
PyFrameObject *
PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject *globals,
      PyObject *locals)
{
    //从 PyThreadState 中获得当前线程的当前执行环境
    PyFrameObject *back = tstate->frame;
    PyFrameObject *f;
    ......
    //创建新的执行环境
    f = PyObject_GC_Resize(PyFrameObject, f, extras);
    ......
    //链接当前执行环境
    f->f_back = back
    f->f_tstate = tstate;
    ......
    return f;
}
```

现在我们发散思维，想象一下 Python 在运行时的某刻，内存中所有参与执行的关键基础对象的布局，如图 8-6 所示：

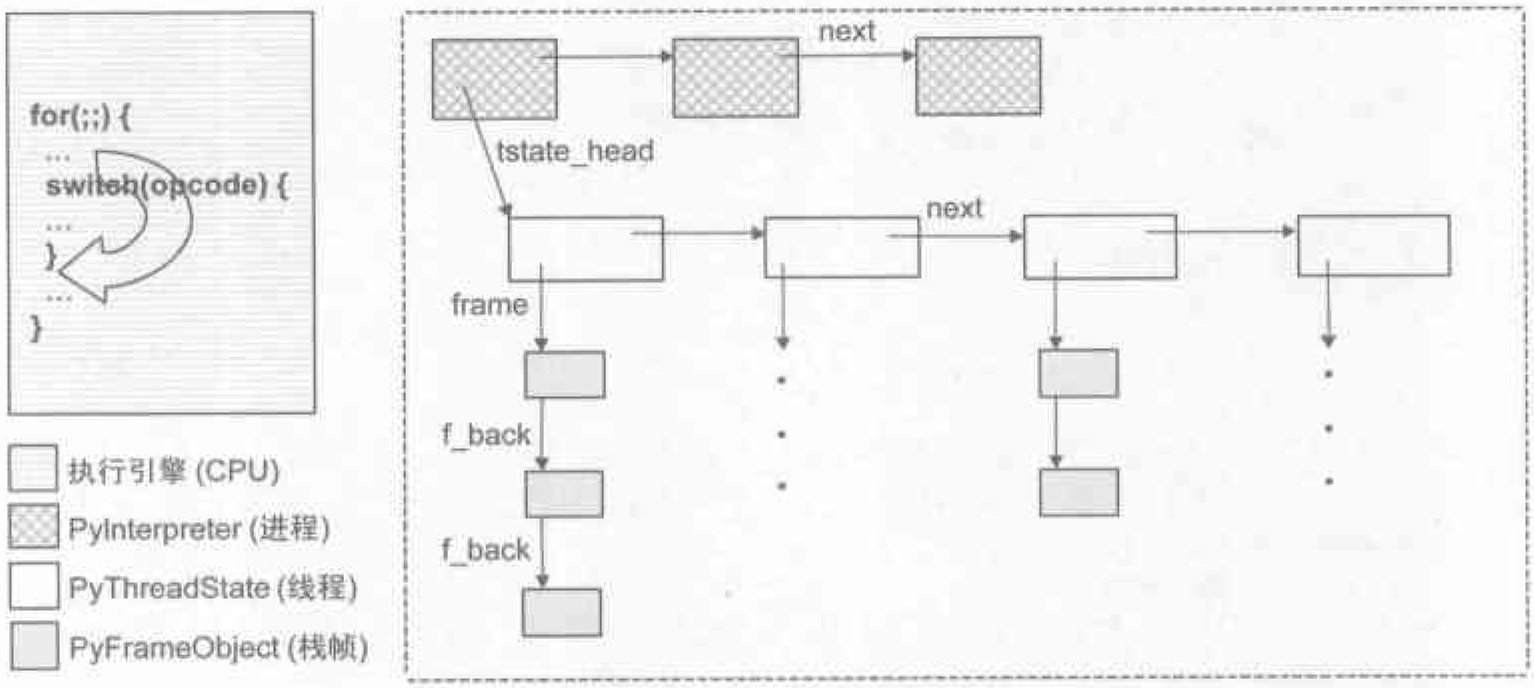



图 8-6 Python 的运行时环境

好了，有了这些基础知识，我们就可以从容面对 Python 虚拟机的一切内容了。

CHAPTER

第 9 章

# Python 虚拟机中的一般表达式

在上一章中，我们已经通过 PyEval_EvalFrameEx 看到了 Python 虚拟机的整体框架，从这章开始，我们将深入到 PyEval_EvalFrameEx 的各个细节当中，深入剖析 Python 的虚拟机。在本章中，我们将剖析 Python 虚拟机是如何完成对 Python 中的一般表达式的执行的。在这里所谓的“一般表达式”包括最基本的对象创建语句，打印语句等。至于 if，while 等表达式，我们将之归类为控制流语句，对于 Python 中控制流的详细剖析，我们将留到下一章。本章将通过对 3 个简单的 Python 源文件的考察，完成对 Python 中一般表达式的剖析。

## 9.1 简单内建对象的创建

还记得在剖析 PyCodeObject 的最后我们所展示 simple_obj.py 吗，我们对 Python 虚拟机的剖析就从这里开始。在 simple_obj.py 中，我们仅仅是创建了一些 Python 中最简单的对象。

```
[simple_obj.py]
i = 1
s = "Python"
d = {}
l = []
```

我们从分析 simple_obj.py 对应的 PyCodeObject 对象中的常量表 co_consts 和符号表 co_names 入手。对于 simple_obj.py，利用我们之前开发的 PycParser 工具解析 simple_obj.py 对应的 pyc 文件，解析的结果如图 9-1 所示。看看 co_consts 和 co_names 中究竟都有些什么：

```
- <consts>
    <int value="1" />
    <internStr index="0" length="6" value="Python" />
    <NoneObject />
  </consts>
- <names>
    <internStr index="1" length="1" value="i" />
    <internStr index="2" length="1" value="s" />
    <internStr index="3" length="1" value="d" />
    <internStr index="4" length="1" value="l" />
  </names>
```

图 9-1 simple_obj.pyc 中的常量表和符号表

图 9-1 中所展示的常量表和符号表是 simple_obj.py 中关于程序运行的重要信息。在随后的剖析中我们可以清楚地看到，这些信息在虚拟机执行字节码指令的过程中具有非常重要的作用。

在前一章中，我们已经看到了 PyEval_EvalFrameEx 中定义了一些在遍历字节码指令序列 co_code 时所必须的宏。其实，在 PyEval_EvalFrameEx 的实现中，出于对效率的考虑，使用了大量的宏，其中的一些宏包括了对栈的各种操作以及对 tuple 元素的访问操作，在执行字节码指令时，会大量使用这些宏。我们来看一看在本章的剖析中需要的一些宏的定义：

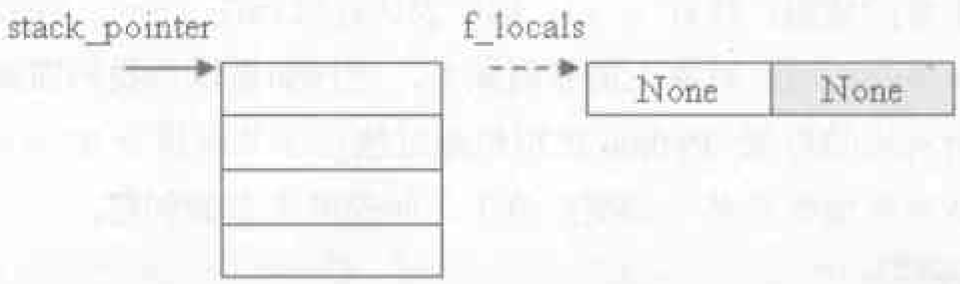
```
[PyEval_EvalFrameEx in ceval.c]
//访问 tuple 中的元素
#define GETITEM(v, i) PyTuple_GET_ITEM((PyTupleObject *)(v), (i))
//调整栈顶指针
#define BASIC_STACKADJ(n)   (stack_pointer += n)
#define STACKADJ(n) BASIC_STACKADJ(n)
//入栈操作
#define BASIC_PUSH(v)   (*stack_pointer++ = (v))
#define PUSH(v)     BASIC_PUSH(v)
//出栈操作
#define BASIC_POP() (*--stack_pointer)
#define POP()       BASIC_POP()
```

在剖析 PyCodeObject 时，我们已经利用 Python 的 dis 工具对 simple_obj.pyc 中的字节码指令进行了解析，这里首先来看一看对第一行 Python 代码的执行：

```
i = 1
0   LOAD_CONST    0  (1)
3   STORE_NAME    0  (i)
```

其中粗体部分那一列表示当前的字节码指令对源文件中的哪个符号或常量进行了操作或产生了影响（下同）。在本章对 Python 虚拟机的剖析中，我们的重点将放在字节码指令将如何影响当前活动的 PyFrameObject 对象中的运行时栈和 local 名字空间(f->f_locals)。字节码指令对符号或常量的操作最终都将反映到运行时栈和 local 名字空间中。首先，我们通过图 9-2 观察一下运行时栈和 local 名字空间的初始情况。在 local 名字空间中，将存储程序执行过程中的局部变量。实际上，local名字空间也就是 Python 虚拟机

的局部变量表。关于它的作用和重要性，在后面的剖析中我们会看得非常清楚。




注意：在以后的阐述中，虚线指针代表 f_locals，实线指针代表 stack_pointer。

图 9-2 初始状态时的运行时栈和 local 名字空间

前面我们已经提到，Python 的整个虚拟机实际上都是由 `PyFrame_EvalFrameEx` 实现的。在剖析虚拟机时，我们展示字节码指令的实现代码的方式与前面不同，不再列出代码所处的文件，而仅仅给出代码是哪个字节码指令的实现代码。默认情况下，所展示的代码都位于 ceval.c 的 `PyFrame_EvalFrameEx` 中，如果有例外，我们会给出代码所在文件的文件名。读者如需对照 Python 源代码，比如对下面列出的 `LOAD_CONST` 指令的实现代码，想对照 Python 源码中源代码，可在 ceval.c 中搜索字符串"`LOAD_CONST`"定位源码位置。

对于第一条字节码指令 `LOAD_CONST`——"`0 LOAD_CONST 0`"，虚拟机的执行动作如下：

```
[LOAD_CONST]
x = GETITEM(consts, oparg);
Py_INCREF(x);
PUSH(x);
```

其中，`GETITEM(consts, oparg)` 显然就是 `GETITEM(consts, 0)`，即 `PyTuple_Get_ITEM(consts, 0)`。`LOAD_CONST` 的意图很明显，就是从 `consts` 中读取序号为 0 的元素，然后将其压入虚拟机的运行时栈中。在 `PyEval_EvalFrameEx` 中可以发现，`consts` 实际上就是 `f->f_code->co_consts`，其中 `f` 是当前活动的 `PyFrameObject` 对象，那么，`consts` 也就是 `PyCodeObject` 对象中的 `co_consts`（在以后的叙述中，我们有时会把这个 `consts` 称为常量表）。

对照图 9-1，我们可以发现，`consts` 中的第 0 个元素是一个整数对象 1，这也是 simple_obj.py 中所创建的第一个对象。`LOAD_CONST` 完成后的情况如图 9-3 所示：

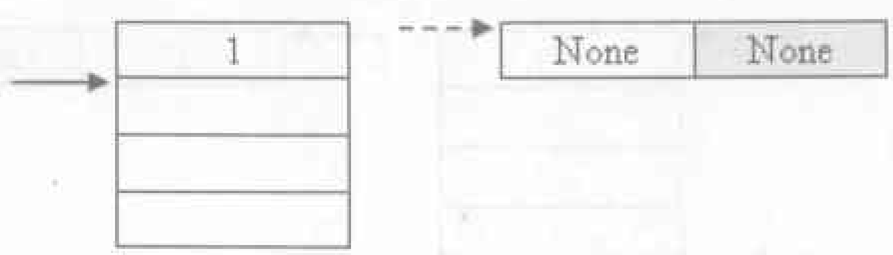



图 9-3 LOAD_CONST 之后的虚拟机状态

第一条字节码指令 LOAD_CONST 只改变了运行时栈，对 local 名字空间没有任何影响。但是按照正常的猜测，执行 i = 1 这个表达式应该在 local 名字空间中创建一个从符号“i”到 PyIntObject 对象 1 的映射关系，这样如果以后我们需要使用符号 i，才能寻找到符号 i 所对应的对象。Python 虚拟机通过执行字节码指令 STORE_NAME 来改变 local 名字空间，从而完成变量名 i 到变量值 1 之间映射关系的创建。

```
[STORE_NAME]
//从符号表中获得符号，其中 oparg = 0
w = GETITEM(names, oparg);
//从运行时栈中获得值
v = POP();
if ((x = f->f_locals) != NULL)
{
    //将（符号，值）的映射关系存储到 local 名字空间中
    if (PyDict_CheckExact(x))
    {
        PyDict_SetItem(x, w, v);
    }
    else
    {
        PyObject_SetItem(x, w, v);
    }
    Py_DECREF(v);
}
```

在这里，我们只考虑 local 名字空间（f->f_locals）确实是 PyDictObject 对象的情况。一般情况下，f->f_locals 都会是一个 PyDictObject 对象。字节码指令“3 STORE_NAME 0”首先根据指令参数从符号表 names 中读取序号为 0 的元素作为变量名，然后将“0 LOAD_CONST 0”指令读取的元素作为变量值，将（变量名，变量值）元素对添加到 f->f_locals 中。与上面看到的 consts 类似，这里的 names 实际上是 f->f_code->co_names（同样与 consts 类似，在以后的叙述中，我们有时将 names 称为符号表）。这里的指令参数是 0，对照图 9-1，我们发现，STORE_NAME 获得的变量名确实是“i”。

现在我们可以很清晰地看到 Python 代码中变量名与变量值在内存中是通过怎样的一种方式捆绑在一起的了。指令“3 STORE_NAME 0”完成后的虚拟机状态如图 9-4 所示。注意，由于在 STORE_NAME 指令的执行过程中，进行了 POP 的动作，所以这时运行时栈中已不存在任何对象了。

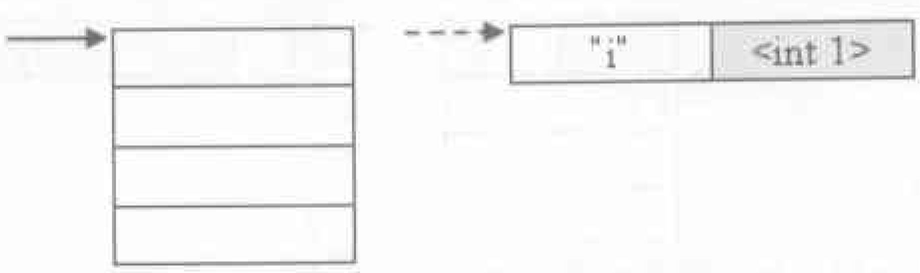



图 9-4 STORE_NAME 之后的虚拟机状态

simple_obj.py 中第 1 行代码执行完毕，第 2 行代码所产生的字节码序列实际上跟第 1 行代码是一样的，只是操作的参数不同了：

```
s = "Python"
6   LOAD_CONST   1  ('Python')
9   STORE_NAME   1  (s)
```

图 9-5 展示了这段字节码序列执行时 Python 虚拟机的运行时栈和 `local` 名字空间的动态变化过程：

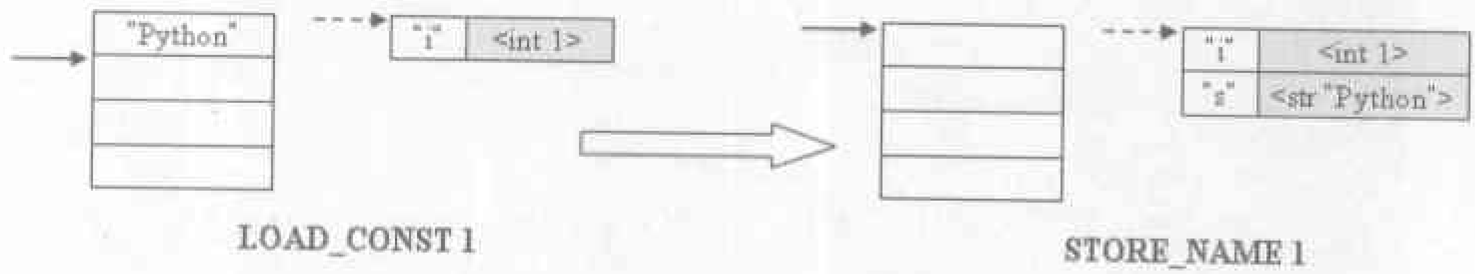



图 9-5 s= “python” 执行过程中虚拟机状态的转变

在 simple_obj.py 的第 3 行，我们见到了一点新鲜的东西，在这里，我们并不是简单地 load 了，而是凭空创建了一个 `PyDictObject` 对象：

```
d = {}
12  BUILD_MAP    0
15  STORE_NAME   2 (d)
```

对于指令“`12 BUILD_MAP 0`”，Python 虚拟机在执行指令时会创建一个空的 `PyDictObject` 对象，并把这个对象压入到运行时栈中：

```
[BUILD_MAP]
x = PyDict_New();
PUSH(x);
```

细心的读者一定发现了，这里有一件很奇怪的事：从 Python 源代码编译出的字节码中，可以发现，`BUILD_MAP` 是一条带有参数的字节码指令，从 opcode.h 中也能证实这一点，但是在这里，我们根本没有看到使用这个参数。可能，这又是“历史遗留”问题☺。

接下来的是 `STORE_NAME` 指令，看到这条指令，实际上我们就可以看到执行完毕后的情形了，如图 9-6 所示：

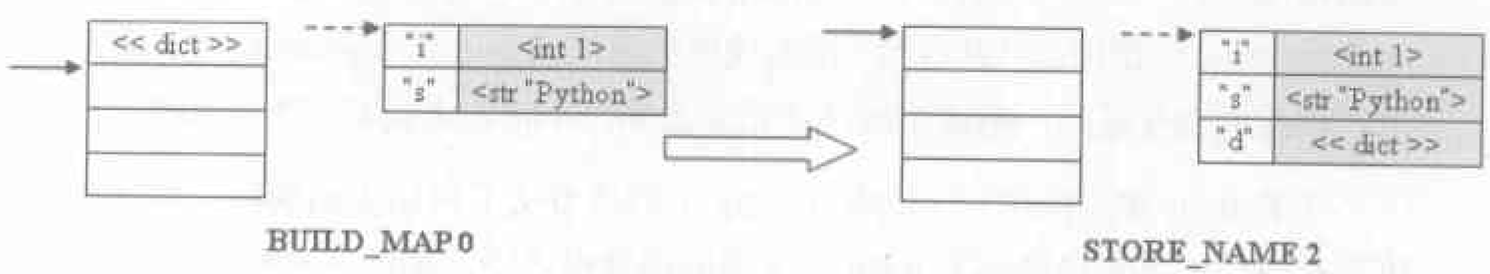



图 9-6 d={}执行过程中虚拟机状态的转变

对于 simple_obj.py 中最后一行 Python 代码，居然编译出了 4 条字节码指令：

```
l = []
18  BUILD_LIST   0
```

```
21   STORE_NAME    3 (1)
24   LOAD_CONST    2 (none)
27   RETURN_VALUE
```

对于指令“18 BUILD_LIST 0”，我们猜想它所执行的操作会与 BUILD_MAP 指令类似，但是，BUILD_LIST 更好，它善待了字节码指令所带的指令参数，真正利用了这个参数，而不是像 BUILD_MAP 那样，仅仅做个摆设：

```
[BUILD_LIST]
x =  PyList_New(oparg);
if (x != NULL)
{
    for (; --oparg >= 0;)
    {
        w = POP();
        PyList_SET_ITEM(x, oparg, w);
    }
    PUSH(x);
}
```

可以推测，如果 Python 源代码中创建的不是一个空的 list，那么在 BUILD_LIST 指令之前一定会有许多 LOAD_CONST 的操作，这将导致有许多对象被压入运行时栈中。在真正执行 BUILD_LIST 指令时，会将这些对象一一从栈中弹出元素，加入到新创建的 PyListObject 对象中，因为这些对象其实就是 list 中的元素。这一点我们后面会详细考察。

在执行了接下来的字节码指令“21 STORE_NAME 3”后，似乎 simple_obj.py 中指定的所有工作都完成了，那最后两条字节码指令的作用呢？

原来 Python 在执行了一段 Code Block 后，一定要返回一些值，这两条字节码指令就是用来返回某些值的：

```
[RETURN_VALUE]
retval = POP();
why = WHY_RETURN;
```

实际的返回值在 retval 中，是从运行时栈中取得的，所以 RETURN_VALUE 前的那条字节码指令“24 LOAD_CONST 2”的作用就很清楚了，它将返回值压入运行时栈中，以供 RETURN_VALUE 使用。可以看到，压入栈中的返回值是一个 NoneObject，实际上什么有价值的东西也没有返回，但这个过场还是要走的，不走，人民是不答应的☺。

在 Python 虚拟机执行 simple_obj.py 中的所有字节码指令的瞬间，运行时栈变空了，所有有用的信息都已经到了 local 名字空间的掌握之中，如图 9-7 所示：

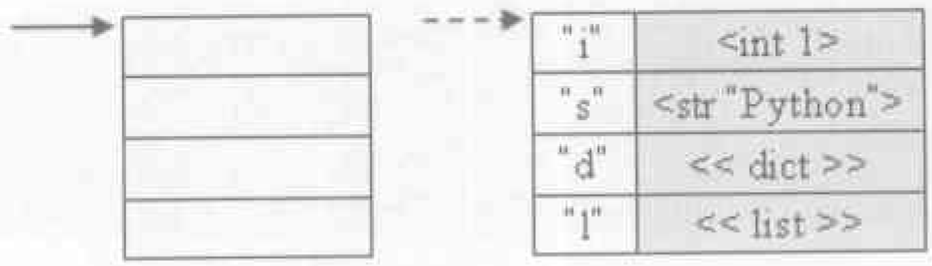



图 9-7 Python 虚拟机结束时的状态

## 9.2 复杂内建对象的创建

前面我们看了 Python 是如何在运行时创建空的 `dict` 对象和空的 `list` 对象的，那么如果是创建非空的 `dict` 和 `list`，Python 的运行时行为又是如何的呢？这个问题很有趣，我们通过 adv_obj.py 来研究：

```
[adv_obj.py]
i = 1
s = "Python"
d = {"1":1, "2":2}
l = [1, 2]
```

对于 adv_obj.py，在 Python 虚拟机运行期间，它所对应的符号表 names（co_names）与 simple_obj.py 应该是一样的，而常量表 consts（co_consts）应该和 simple_obj.py 是不同的。图 9-8 显示了与 adv_obj.py 对应的 `co_consts` 和 `co_names`：

```
- <consts>
    <int value="1" />
    <internStr index="0" length="6" value="Python" />
    <internStr index="1" length="1" value="1" />
    <int value="2" />
    <internStr index="2" length="1" value="2" />
    <NoneObject />
  </consts>
- <names>
    <internStr index="3" length="1" value="i" />
    <internStr index="4" length="1" value="s" />
    <internStr index="5" length="1" value="d" />
    <internStr index="6" length="1" value="l" />
  </names>
```

图 9-8 adv_obj.pyc 中的符号表和常量表

在编译得到的字节码指令序列中，前两行 Python 代码的字节码序列都是相同的，我们不再考察。而在创建非空的 `dict` 时，字节码序列与 simple_obj.py 中的不同了：

```
d = {"1":1, "2":2}
12   BUILD_MAP    0
15   DUP_TOP
16   LOAD_CONST   0 (1)
19   ROT_TWO
20   LOAD_CONST   2 ('1')
23   STORE_SUBSCR
```

```
24   DUP_TOP
25   LOAD_CONST    3 (2)
28   ROT_TWO
29   LOAD_CONST    4 ('2')
32   STORE_SUBSCR
33   STORE_NAME    2 (d)
```

指令“12 BUILD_MAP 0”我们已经知道它的作用了，接下来的指令“15 DUP_TOP”却是一条全新的指令。对于这条指令，Python 虚拟机会进行如下的动作：

```
[DUP_TOP]
v = TOP();
Py_INCREF(v);
PUSH(v);
```

需要注意的是，DUP_TOP 指令不只是增加了栈顶元素的引用计数，还将栈顶元素又一次压入栈中。如此看上去，DUP 似乎是 duplicate 的缩写。这一动作非常诡异，不过马上我们就能看到这个动作的用意。由于在 DUP_TOP 指令之前是一条“12 BUILD_MAP 0”指令，所以会将创建的 PyDictObject 对象的引用计数增加 1，并再次压入该 PyDictObject 对象。

紧接着的“16 LOAD_CONST 0”指令会从 consts 中将需要插入到 PyDictObject 对象中的第一个元素对的值（<int 1>）读取出来，压入运行时栈中，前 3 条字节码指令完成后的情形如图 9-9 所示：

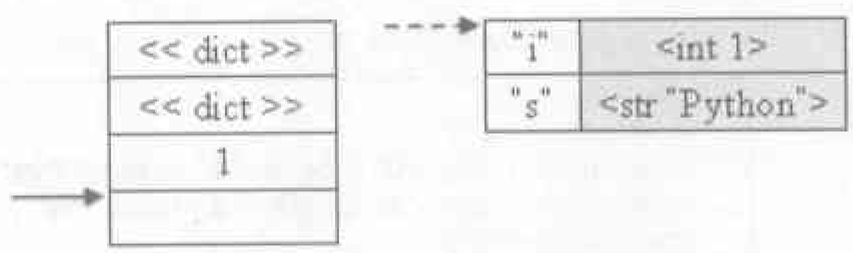



图 9-9 ROT_TWO 之前虚拟机的状态

Python 虚拟机在接下来对指令“19 ROT_TWO”的执行过程中，会做一些奇怪的动作：

```
[ROT_TWO]
v = TOP();
w = SECOND();
SET_TOP(w);
SET_SECOND(v);
```

其中的 SET_TOP 等宏也是在 PyEval_EvalFrameEx 中定义的：

```
[ceval.c]
#define TOP()          (stack_pointer[-1])
#define SECOND()       (stack_pointer[-2])
#define SET_TOP(v)     (stack_pointer[-1] = (v))
#define SET_SECOND(v)    (stack_pointer[-2] = (v))
```

仔细观察后我们发现，其实 ROT_TWO 所做的就是将栈顶的两个元素进行对调。ROT_TWO 指令完成后的情形如图 9-10 所示。

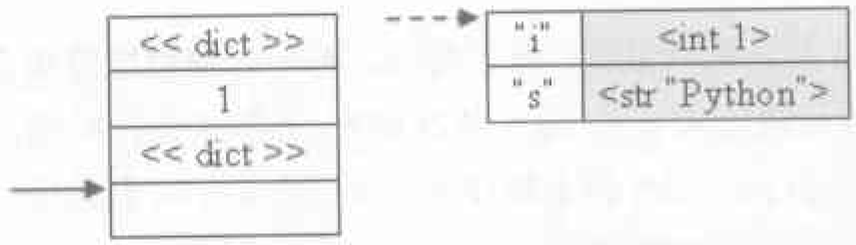



图 9-10 ROT_TWO 之后虚拟机的状态

随后的一条“20 LOAD_CONST 2”指令会从 consts 中将需要插入到 PyDictObject 对象中的第一个元素对的键（<string "1">）读取出来。而紧随的“23 STORE_SUBSCR”指令会将（键，值）对插入到 PyDictObject 对象中去：

```
[STORE_SUBSCR]
w = TOP();      // "1"
v = SECOND(); // dict object
u = THIRD();  // 1
STACKADJ(-3);
//v[w] = u, 即 dict["1"] = 1
PyObject_SetItem(v, w, u);
Py_DECREF(u);
Py_DECREF(v);
Py_DECREF(w);
```

随着 STACKADJ 的执行，栈顶指针回退了 3 格，所以 STORE_SUBSCR 指令执行完后，运行时栈里又只剩下了最初由 BUILD_MAP 创建的 PyDictObject 对象。到这里，Python 虚拟机就完成了将一个元素对插入到 PyDictObject 的操作。剩下的字节码序列以（LOAD_CONST、ROT_TWO、LOAD_CONST、STORE_SUBSCR）4 个字节码为一组，每组的目的是重复上面我们所看到的动作，不断地将后续元素对插入到 PyDictObject 对象中去。当所有的元素对的插入动作都完成之后，将由我们的老朋友“33 STORE_NAME 2”指令最终把这个 PyDictObject 对象添加到 local 名字空间中去,并为其关联一个名为“d”的符号。当所有的字节码指令执行完成之后，情形如图 9-11 所示：

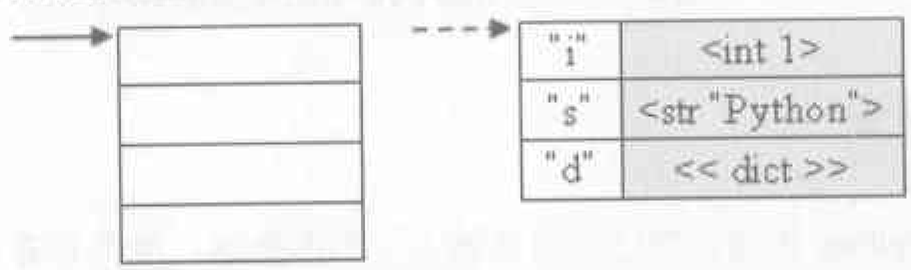



图 9-11 创建 PyDictObject 对象之后的虚拟机状态

在成功创建了非空 PyDictObject 对象之后，Python 虚拟机还会创建一个非空的 PyListObject 对象：

```
l = [1, 2]
36  LOAD_CONST   0 (1)
39  LOAD_CONST   3 (2)
42  BUILD_LIST   2
45  STORE_NAME   3 (l)
```

对照图 9-8 所示的常量表，可以看到，Python 虚拟机确实会首先将应该填入 PyListObject 对象的元素先从 consts 中依次读入，并压入运行时栈。然后如之前描述的，Python 虚拟机在通过 BUILD_LIST 指令创建 PyListObject 对象之后，会从运行时栈中依次将对象读出，从后至前地将它们填入 PyListObject 对象中。

# 9.3 其他一般表达式

到了这里，对于一般的声明语句（或者说常量赋值语句），我们都已经了如指掌。下面，我们通过研究如下所示的 normal.py，考察变量赋值、变量运算及最基本的 print 操作，完成对一般 Python 语句的剖析工作：

```
[normal.py]
a = 5
b = a
c = a + b
print c
```

在 Python 编译器对 normal.py 的编译成功结束之后，其对应的 PyCodeObject 中的 co_consts 和 co_names 如图 9-12 所示：

```
- <consts>
    <int value="5" />
    <NoneObject />
  </consts>
- <names>
    <internStr index="0" length="1" value="a" />
    <internStr index="1" length="1" value="b" />
    <internStr index="2" length="1" value="c" />
  </names>
```

图 9-12 normal.pyc 中的常量表和符号表

## 9.3.1 符号搜索

第 1 行 Python 代码我们已经很熟悉了，不再赘述。现在看看第 2 行 Python 代码，变量之间的赋值运算：

```
b = a
0   LOAD_NAME    0  (a)
3   STORE_NAME   1  (b)
```

现在，对于指令“3 STORE_NAME 1”的作用我们已清楚。而对于指令“0 LOAD_NAME 0”，我们还是第一次遇到（见代码清单 9-1）。

代码清单 9-1

```
[LOAD_NAME（有删节）]
        //获得变量名
        w = GETITEM(names, oparg);
        //[1] ：在local名字空间中查找变量名对应的变量值
        v = f->f_locals;
        x = PyDict_GetItem(v, w);
        Py_XINCREF(x);
        if (x == NULL) {
            //[2] : 在glocal名字空间中查找变量名对应的变量值
            x = PyDict_GetItem(f->f_globals, w);
            if (x == NULL) {
                //[3] : 在builtin名字空间中查找变量名对应的变量值
                x = PyDict_GetItem(f->f_builtins, w);
                if (x == NULL) {
                    //[4] : 查找变量名失败，抛出异常
                    format_exc_check_arg(PyExc_NameError,NAME_ERROR_MSG ,w);
                    break;
                }
            }
            Py_INCREF(x);
        }
        PUSH(x);
```

STORE_NAME 指令非常简单，但没想到看上去与它成逆运算的 LOAD_NAME 指令却会如此复杂。面对 LOAD_NAME 指令，首先，Python 虚拟机会从符号表 names 中抽取出第 0 个元素，参考图 9-12 所示的 normal.py 编译结果，可知它是一个 PyStringObject 对象“a”。然后 Python 虚拟机会在当前活动 PyFrameObject 对象中所维护的多个名字空间中进行一系列的搜索动作，如代码清单 9-1 中的[1]、[2]、[3]所示：

[1] 在 local 名字空间 f_locals 中搜索符号“a”；

[2] 如果 f_locals 中没有符号“a”，则在 global 名字空间 f_globals 中搜索“a”；

[3] 如果 f_globals 中没有符号“a”，则在 Python 的 builtin 名字空间 f_builtins 中搜索“a”。

如果搜索到了与符号“a”对应的元素，那么就将该元素压入运行时栈中。在 normal.py 这个例子中，第 1 条 Python 代码的执行会在 f_locals 中插入（“a”, 5）的元素对，所以这里会向运行时栈中压入一个 PyIntObject 对象 5。

如果到了最后还搜索不到符号“a”，那么表示有错误发生，程序引用了一个不存在的符号。那么 Python 虚拟机的执行流程最终会到达代码清单 9-1 的[4]处，抛出异常，终止 Python 虚拟机的运行。

这样的行为正是 Python 官方文档中所描述的变量的搜索会沿着局部作用域（local 名字空间）、全局作用域（global 名字空间）、内建作用域（builtin 名字空间）依次上

溯，直至搜索成功或全部搜完 3 个作用域，也就是我们之前提到的 LGB 规则。为了更加形象地展示这一符号的搜索过程，我们采用在剖析 Python 对象机制时，修改 Python 源代码的方法，然后在运行时实时地观察这一过程。

我们在代码清单 9-1 的[1]、[2]、[3]、[4]处各添加一段输出信息，这里我们列出在代码清单 9-1 的[1]处添加的输出代码：

```
if(strcmp(PyString_AsString(w), "PythonVM") == 0)
{
    PyObject* target = PySys_GetObject("stdout");
    char temp[256] = {0};
    sprintf(temp, "[LOAD_NAME] : Search PyStrinObject %s in local name
      space...%s\n", PyString_AsString(w), x==NULL?"False":"Success");
    PyFile_WriteString(temp, target);
}
```

由于我们在这里使用 IDLE 来观察输出信息，所以修改了代码之后，我们需要自行编译出 python25.dll 文件。注意，这里一定要在 release 模式下编译。因为 IDLE 程序本身会使用 Python 标准库中的 socket module，所以 IDLE 初始化时会进行 `import socket` 的动作。如果在 debug 模式下编译得到 python25_d.dll，那么在 `import socket` 时，就会寻找 _socket_d.pyd，这意味着我们还得继续编译 socket 工程。所以我们选择在 release 模式下编译 python25.dll，因为当你在机器上安装 Python 后，_socket.pyd 已经在 Python 安装目录下的 dll 子目录中存在了。

在编译完成之后，我们将 python25.dll 拷贝到 Python 的安装目录下。由于 Python 安装时是将 python25.dll 拷贝到了 system32 目录下，所以对于 python25.dll，直接拷贝就可以了。图 9-13 显示了一次失败的符号搜索过程。

```
>>> print PythonVM
[LOAD_NAME] : Search PyStrinObject PythonVM in local name space... False
[LOAD_NAME] : Search PyStrinObject PythonVM in global name space... False
[LOAD_NAME] : Search PyStrinObject PythonVM in builtin name space... False
[LOAD_NAME] : Search failed, throw exception

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print PythonVM
NameError: name 'PythonVM' is not defined
```

图 9-13 搜索符号“PythonVM”失败

现在很清楚了，在“`b = a`”执行完成后，Python 虚拟机的状态如图 9-14 所示：

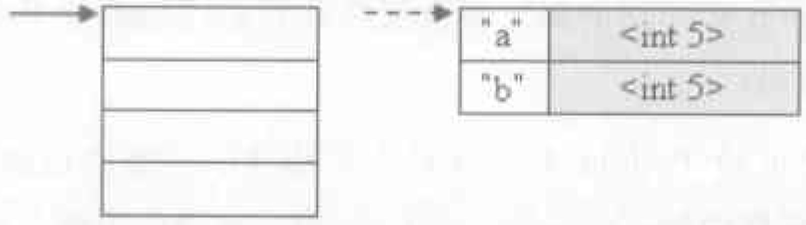



图 9-14 b=a 执行之后的虚拟机状态

## 9.3.2 数值运算

从前面我们对 PyIntObject 对象的分析可以知道，这两个 5 实际上指向内存中的同一个 PyIntObject 对象。这里在对字节码指令的分析中，我们再次看到了这一结论。现在 a，b 都是合法而有效的变量了，它们的结合就变得很有趣了：

```
c = a + b
12   LOAD_NAME   0  (a)
15   LOAD_NAME   1  (b)
18   BINARY_ADD
19   STORE_NAME   2 (c)
```

Python 虚拟机首先会通过两条 LOAD_NAME 指令将变量名 a 和 b 所对应的变量值从 local 名字空间中读取出来，压入运行时栈，然后通过 BINARY_ADD 指令进行加法运算，计算两个变量的和。假设加法运算的结果为 num，那么 Python 虚拟机在获得结果 num 之后，会通过指令 STORE_NAME 将（"c"，num）元素对插入到 local 名字空间中。这里展示的是变量之间的加法运算，当然，这里的加法运算实际上完全可以替换成减法运算、乘法运算，有兴趣的读者可以自行考察一下其他运算所对应的字节码指令。

现在我们感兴趣的是那个从两个现有对象创造出新的对象的"18 BINARY_ADD"指令，如代码清单 9-2 所示。

**代码清单 9-2**

```
[BINARY_ADD]
        w = POP();
        v = TOP();
        if (PyInt_CheckExact(v) && PyInt_CheckExact(w)) {
            //[1]: PyIntObject 对象相加的快速通道
            register long a, b, i;
            a = PyInt_AS_LONG(v);
            b = PyInt_AS_LONG(w);
            i = a + b;
            //[2]: 如果加法运算溢出，转向慢速通道
            if ((i^a) < 0 && (i^b) < 0)
                goto slow_add;
            x = PyInt_FromLong(i);
        }
        //[3]: PyStringObject 对象相加的快速通道
        else if (PyString_CheckExact(v) && PyString_CheckExact(w)) {
            x = string_concatenate(v, w, f, next_instr);
            goto skip_decref_vx;
        }
        else {
//[4]: 一般对象相加的慢速通道
slow_add:
            x = PyNumber_Add(v, w);
        }
        Py_DECREF(v);
skip_decref_vx:
```

```
        Py_DECREF(w);
        SET_TOP(x);
        break;
```

想不到一个简简单单的"加法"竟然会有如此复杂的实现过程。在 BINARY_ADD 指令的实现中，Python 虚拟机为对象之间的加法运算建立了两条通道，一条快速通道，一条慢速通道。快速通道仅仅是为 PyIntObject 对象和 PyStringObject 对象准备的，而其他的对象都必须通过慢速通道完成加法运算。

在代码清单 9-2 的[1]处可以看到，如果参与运算的两个对象都是 PyIntObject 对象，会直接将 PyIntObject 中的 value 提取出来相加，然后根据相加的结果创建新的 PyIntObject 对象作为结果返回；同样，在代码清单 9-2 的[3]处，如果是 PyStringObject 对象之间相加，Python 虚拟机也会选择 string_concatenate 以加快速度。[1]和[3]处就是 Python 虚拟机所提供的两条快速通道。

如果参与运算的对象在这两种有加速机制的情况之外，那它们只能通过慢速通道 PyNumber_Add 完成加法运算。在 PyNumber_Add 中，Python 虚拟机会进行大量的类型判断，寻找与对象相对应的加法操作函数等额外工作，速度会比前两种加速机制慢很多。一般来说，Python 虚拟机在 PyNumber_Add 中会首先检查参与运算的对象的类型对象，检查 PyNumberMethods 中的 nb_add 能否完成在 v 和 w 上的加法运算；如果不能，还会检查 PySequenceMethods 中的 sq_concat 能否完成，如果都不能，Python 虚拟机也是有心杀敌，无力回天了，那也只好报告错误了。

值得注意的是，虽然 Python 虚拟机为 PyIntObject 对象准备了快速通道，但是当加法运算的结果发生溢出时，Python 虚拟机会放弃快速通道计算的结果，转向慢速通道。因为快速通道只能产生另一个 PyIntObject 对象作为加法运算的结果，而当溢出发生时，就意味着我们需要一个 PyLongObject 对象作为结果了，这个 PyLongObject 对象的创建只能在慢速通道中完成。因为在慢速通道中，加法运算会委托给 PyIntObject 对象的类型对象中所定义的加法操作，在那里会处理溢出的情况。为了实时地看到这一过程，我们像前面一样，在 BINARY_ADD 指令的实现代码中加入输出信息，重新编译 python25.dll，利用 IDLE 来观察输出结果。输出的结果如图 9-15 所示：

```
>>> a = -1
>>> b = -2
>>> print a+b
[BINARY_ADD] : "-1+-2" in quick channel...  success
-3
>>> a = 0x6fffffff
>>> b = a+a
[BINARY_ADD] : "1879048191+1879048191" in quick channel...  overflow!!
[BINARY_ADD] : "1879048191+1879048191" switch to slow channel...
>>> print b
3758096382
>>> type(b)
<type 'long'>
```

图 9-15 加法指令的两条通道

从两条加法通道的结构我们可以看出，Python2.5 实际上假设了用户在使用 Python 时，通过操作符“+”大量进行的加法操作是整数的加法和字符串的连接。其实如果你的程序中涉及了大量的浮点运算，完全可以修改 BINARY_ADD 的代码，为浮点加法运算建立快速通道。实际上，对于加、减、乘、除这些操作，你都可以根据自己程序的实际情况对 Python 的代码进行改动，这样的改动应该对提升程序运行效率有很大的好处。

## 9.3.3 信息输出

最后来看一看 print 的动作，前面分析的字节码指令都是 Python 自身的动作，这个 print 则是 Python 与用户的交互，也是 Python 中最常用和易用的用户交互方式：

```
print c
22   LOAD_NAME    2 (c)
25   PRINT_ITEM
26   PRINT_NEWLINE
```

如果想要输出什么对象，那么一定先要获得它，与 print 语句相对应的第一条字节码指令的任务就是获取需要输出的对象。Python 虚拟机通过指令“22 LOAD_NAME 2”从 local 名字空间中将加法运算的结果 c 读取出来，压入运行时栈，然后通过指令“25 PRINT_ITEM”完成输出的魔法：

```
[PRINT_ITEM]
        v = POP(); //获得待输出对象
        if (stream == NULL || stream == Py_None) {
            w = PySys_GetObject("stdout");
        }
        Py_XINCREF(w);
        if (w != NULL && PyFile_SoftSpace(w, 0))
            err = PyFile_WriteString(" ", w);
        if (err == 0)
            err = PyFile_WriteObject(v, w, Py_PRINT_RAW);
        ……
         stream = NULL;
```

实际上输出的时候会进行很多动作，但是在这里，我们只考虑其大致的流程。Python 虚拟机首先会将待输出的对象从运行时栈中取出，然后对一个名为 stream 的东西进行判断，如果 stream 为 NULL，则将 w 设为标准输出流。这个 stream 是什么呢？它实际上也是一个 PyObject 对象：

```
PyObject *stream = NULL;    /* for PRINT opcodes */
```

如果输出的时候，是通过如下的 Python 代码：print >> file, "str"。那么所产生的字节码序列中，在 PRINT_ITEM 之前还有一条字节码指令——PRINT_ITEM_TO：

```
[PRINT_ITEM_TO]
w = stream = POP();
```

显然，这时在运行时栈中，在待输出对象之前，还会有一个对象，即输出的目标。在执行 PRINT_ITEM_TO 时，输出的目标就赋给了 stream，同时也赋给了 w。所以实际上 stream 是作为一个判断条件来使用的，真正使用的输出目标是 w。要多次使用这一个 stream 的原因是变量 w 在别的字节码指令的实现中可能还会使用到，所以无法通过判断 w 是否为 NULL 来确定是否需要输出到标准输出流。可以看到，在 PRINT_ITEM 最后，又将 stream 设为了 NULL，为下次输出时的判断做准备。

图 9-16 给出了在 PRINT_ITEM 中输出目标的两种情况：

```
>>> print 1
w is a file with name : '<stdout>'
1
>>> f = open('demo.txt', 'w')
>>> print >> f, 1
w is a file with name : 'demo.txt'
```

图 9-16 PRINT_ITEM 中 w 所代表的两种输出目标

在获得了输出的目标和待输出的对象后，PRINT_ITEM 将通过 PyFile_WriteObject -> PyObject_Print -> internal_print 的调用序列最终调用 v->ob_type->tp_print，等待输出对象自身所携带的输出函数进行输出。如果对象没有定义 tp_print，那么它会先调用 tp_str 或 tp_repr 获得对象的字符串表示形式，然后将字符串输出。如果连这最后的救命稻草也失败了，Python 也无能为力了，只好返回失败信息。

CHAPTER

第 10 章

# Python 虚拟机中的控制流

在上一章，我们剖析了 Python 虚拟机中的一般表达式的实现。在剖析一般表达式时，我们所考察的程序的流程都是顺序执行的，在执行的过程中没有任何变化。在几年前风靡全球的好莱坞巨片《Matrix》中，片中人物之间曾有诸多关于"控制"、"选择"等概念的对话。一个只会顺序执行的程序是没有什么趣味的，所有的现代编程语言中，都提供了控制程序流程的语义元素，在 C 中甚至还有像 goto 这样的瞬间超距移动语义。在本章中，我们将剖析 Python 中所提供的所有的流程控制手段，其中包含了异常机制。

## 10.1 Python 虚拟机中的 if 控制流

### 10.1.1 研究对象——if_control.py

在所有的编程语言中，if 控制流是最简单也是最常用的流程控制语句。在这一节，我们将通过对如下所示的 if_control.py 的研究来深入剖析 Python 对选择控制语句的实现。

```
[if_control.py]
a = 1
if a > 10:
    print "a > 10"
elif a <= -2:
    print "a <= -2"
elif a != 1:
    print "a != 1"
elif a == 1:
    print "a == 1"
else:
    print "Unknown a"
```

在 if_control.py 中，我们除了剖析 Python 虚拟机中对 if 控制流的实现之外，还会剖析 Python 虚拟机中对比较操作的实现，所以在 if_control.py 中，我们列出了 Python 所支

持的几乎所有的判断操作。

根据我们现在对 Python 编译器的工作过程的了解，我们知道，在 Python 编译器完成对 if_control.py 的编译之后，只会得到一个 PyCodeObject 对象。这个 PyCodeObject 对象中所包含的与 if_control.py 所对应的常量表（co_consts）和符号表（co_names）如图 10-1 所示：

```
<consts>
  <int value="1"/>
  <int value="10"/>
  <str length="6" value="a > -1"/>
  <int value="-2"/>
  <str length="7" value="a <= -2"/>
  <str length="6" value="a != 1"/>
  <str length="6" value="a == 1"/>
  <str length="9" value="Unknown a"/>
  <NoneObject/>
</consts>
<names>
  <internStr index="0" length="1" value="a"/>
</names>
```

图 10-1 if_control.pyc 中的常量表和符号表

在 if_control.py 中，第一行代码是我们已经非常熟悉的了。第一行代码执行的结果会在当前活动的 PyFrameObject 对象的 local 名字空间内添加一个名为“a”的符号，且其关联着一个 PyIntObject 对象。

对于其后的 if 控制流部分，编译所得的字节码指令序列如下所示：

```
if a > 10:
6 LOAD_NAME     0 (a)
9 LOAD_CONST    1 (10)
12 COMPARE_OP   4 (>)
15 JUMP_IF_FALSE 9 (to 27)
18 POP_TOP
   print "a > -1"
19 LOAD_CONST   2 ('a>-1')
22 PRINT_ITEM
23 PRINT_NEWLINE
24 JUMP_FORWARD 72 (to 99)
27 POP_TOP
elif a <= -2:
28 LOAD_NAME    0 (a)
31 LOAD_CONST   3 (-1)
34 COMPARE_OP   1 (<=)
37 JUMP_IF_FALSE 9 (to 49)
40 POP_TOP
   print "a <= -2"
41 LOAD_CONST   4 ('a<=-2')
44 PRINT_ITEM
45 PRINT_NEWLINE
46 JUMP_FORWARD 50 (to 99)
49 POP_TOP
elif a != 1:
50 LOAD_NAME    0 (a)
53 LOAD_CONST   0 (1)
56 COMPARE_OP   3 (!=)
59 JUMP_IF_FALSE 9 (to 71)
62 POP_TOP
   print "a != 1"
63 LOAD_CONST   5 ('a!=1')
66 PRINT_ITEM
67 PRINT_NEWLINE
68 JUMP_FORWARD 28 (to 99)
71 POP_TOP
elif a == 1:
72 LOAD_NAME    0 (a)
75 LOAD_CONST   0 (1)
78 COMPARE_OP   2 (==)
81 JUMP_IF_FALSE 9 (to 93)
84 POP_TOP
   print "a == 1"
85 LOAD_CONST   6 ('a==1')
88 PRINT_ITEM
89 PRINT_NEWLINE
90 JUMP_FORWARD 6 (to 99)
93 POP_TOP
else:
   print "Unknown a"
94 LOAD_CONST   7
97 PRINT_ITEM
98 PRINT_NEWLINE
99 LOAD_CONST 8 (None)
102 RETURN_VALUE
```

注意：字节码指令序列的顺序是从左到右，从上到下。

## 10.1.2 比较操作

与其他的程序设计语言相同，Python 中的 if 控制语句也是通过一个判断来控制程序的流程的。在 if 语句处，根据判断结果的不同，程序将选择向左走还是向右走。选择不同的道路，就有可能经历不同的计算过程，最终导致不同的计算结果。所以，判断操作对于程序流程的控制至关重要。其实不仅仅是对于 if 控制语句，对于其他的流程控制语句，判断也是关键的一环，这一点我们将在以后看到。

在进行程序设计时，判断主要就是对两个对象进行比较，判断的结果也就是比较的结果。所以在本章的描述中，判断和比较这两个概念是可以互相替换的。现在，我们就以比较操作为切入点，开始进入对 if 控制流的剖析。

仔细观察 if_control.py 中的各个判断分支语句，我们发现它们经 Python 编译器编译后都呈现出同样的字节码指令序列结构：

- 执行 LOAD_NAME 指令，从 local 名字空间中获得变量名 a 所对应的变量值；
- 执行 LOAD_CONST 指令，从常量表 consts 中读取参与该分支判断操作的常量对象；
- 执行 COMPARE_OP 指令，对前面两条指令取得的变量值和常量对象进行比较操作；
- 执行某一条 JUMP_*指令，根据 COMPARE_OP 指令的运行结果进行字节码指令的跳跃。

可以看到，不同的分支判断语句编译后都是这样相同的指令序列结构，但是不同的分支判断语句进行的是不同的比较操作。显然在相同的指令序列结构中，应该有某个细节的地方有所不同。通过观察，我们发现不同的分支判断在调用 COMPARE_OP 指令时，传入的指令参数确实是不同的。所以，COMPARE_OP 指令的指令参数实际上也就是区分这些不同分支判断的关键所在。实际上，COMPARE_OP 的不同指令参数对应了不同的比较操作。这些不同的比较操作在 object.h 中定义：

```
[object.h]
/* Rich comparison opcodes */
#define Py_LT 0
#define Py_LE 1
#define Py_EQ 2
#define Py_NE 3
#define Py_GT 4
#define Py_GE 5

[opcode.h]
enum cmp_op {PyCmp_LT=Py_LT, PyCmp_LE=Py_LE, PyCmp_EQ=Py_EQ,
PyCmp_NE=Py_NE, PyCmp_GT=Py_GT, PyCmp_GE=Py_GE,PyCmp_IN, PyCmp_NOT_IN,
PyCmp_IS, PyCmp_IS_NOT, PyCmp_EXC_MATCH, PyCmp_BAD};
```

从 if_control.py 编译的结果可以看出，判断 `a > 10` 对应的 COMPARE_OP 的指令参数是 4，即 PyCmp_GT，正是 Greater 的缩写；而 PyCmp_LE 是 less and equal 的缩写，所以判断 `a <= -2` 对应的 COMPARE_OP 指令的指令参数正是 1。

### 10.1.2.1 COMPARE_OP 指令

了解了 Python 中不同的比较操作所对应的指令参数之后，我们可以深入到 Python 虚拟机中通用的用于比较操作的字节码指令——COMPARE_OP 中了（见代码清单 10-1）。

**代码清单 10-1**

```
[COMPARE_OP]
        w = POP();
        v = TOP();
        //[1]: PyIntObject 对象的快速通道
        if (PyInt_CheckExact(w) && PyInt_CheckExact(v)) {
            register long a, b;
            register int res;
            a = PyInt_AS_LONG(v);
            b = PyInt_AS_LONG(w);
            //根据字节码指令的指令参数选择不同的比较操作
            switch (oparg) {
            case PyCmp_LT: res = a <  b; break;
            case PyCmp_LE: res = a <= b; break;
            case PyCmp_EQ: res = a == b; break;
            case PyCmp_NE: res = a != b; break;
            case PyCmp_GT: res = a >  b; break;
            case PyCmp_GE: res = a >= b; break;
            case PyCmp_IS: res = v == w; break;
            case PyCmp_IS_NOT: res = v != w; break;
            default: goto slow_compare;
            }
            x = res ? Py_True : Py_False;
            Py_INCREF(x);
        }
        else {
    //[2]: 一般对象的慢速通道
    slow_compare:
            x = cmp_outcome(oparg, v, w);
        }
        Py_DECREF(v);
        Py_DECREF(w);
        //将比较结果压入到运行时栈中
        SET_TOP(x);
        if (x == NULL) break;
        PREDICT(JUMP_IF_FALSE);
        PREDICT(JUMP_IF_TRUE);
```

和我们在上一章所看到的 BINARY_ADD 指令一样，Python 虚拟机在 COMPARE_OP 指令的实现中为 PyIntObject 对象建立了快速通道。如果参与比较操作的两个对象都是

PyIntObject 对象，那么直接取得 PyIntObject 对象中维护的整数值进行比较即可。从代码清单 10-1 中可以清晰地看到，Python 正是通过 COMPARE_OP 指令的不同指令参数来选择不同的比较操作的。

如果参与比较的两个对象不全是 PyIntObject 对象，很不幸，只能进入 Python 虚拟机为比较操作准备的慢速通道，调用 cmp_outcome 进行常规的比较操作。这个常规的比较操作与 PyIntObject 对象的快速通道相比，执行效率真是有天壤之别，马上我们就可以看到了：

```
[ceval.c]
static PyObject* cmp_outcome(int op, register PyObject *v, register PyObject *w)
{
    int res = 0;
    switch (op) {
    case PyCmp_IS:
        res = (v == w);
        break;
    case PyCmp_IS_NOT:
        res = (v != w);
        break;
    case PyCmp_IN:
        res = PySequence_Contains(w, v);
        if (res < 0)
            return NULL;
        break;
    case PyCmp_NOT_IN:
        res = PySequence_Contains(w, v);
        if (res < 0)
            return NULL;
        res = !res;
        break;
    case PyCmp_EXC_MATCH:
        res = PyErr_GivenExceptionMatches(v, w);
        break;
    default:
        return PyObject_RichCompare(v, w, op);
    }
    v = res ? Py_True : Py_False;
    Py_INCREF(v);
    return v;
}
```

在 cmp_outcome 的实现代码中，实际上透露了另一个信息，即 Python 的 COMPARE_OP 指令不仅管辖着两个对象之间比较操作，而且还覆盖了对象与集合之间关系的判断操作。比如下面的 Python 代码：

```
list = [1, 2, 3, 4]
if 1 in list:
# LOAD_CONST    0 (1)
# LOAD_NAME     0 (list)
# COMPARE_OP    6 (in)
# JUMP_IF_FALSE
```

```
# POP_TOP
  print "Found"
```

在第 2 行代码编译后的字节码指令序列中我们发现，其中的 in 操作符最终也会被编译为 COMPARE_OP 指令，而指令的参数则是 PyCmp_IN。所以，cmp_outcome 实际上主要是处理这些广义上的比较操作，甚至还包揽了 is 操作符的实现。对于 PyCom_IN 操作，com_outcom 会委托给 PySequence_Containes 来判断在序列对象 w 中是否存在对象 v；而对于通常意义上的两个对象之间的大小关系的比较操作，com_outcome 委托给 PyObject_RichCompare 进行。

在 PyObject_RichCompare 中，首先会确保执行的比较操作在 Py_LT 和 Py_GE 之间，即常规意义上的比较操作。如果进行比较操作的两个对象类型相同，且这两个对象不是用户自定义的类的实例对象，那么首先会选择对象对应的 PyTypeObject 对象中所定义的 tp_richcompare 操作；如果类型对象没有定义这个动作，就选择类型对象中定义的 tp_compare 操作。在 Python 中，无论是 Python 内建对象，还是用户自定义的类的实例对象，其比较操作都是在各自对应的类型对象中的 tp_richcompare 或 tp_compare 中定义的。如果这两个操作都没有成功，那么 Python 还不死心，还会调用 do_richcmp 进行垂死挣扎☺。

到了这里，如果继续剖析，我们就将陷入 Python 中复杂的对象比较体系了，这里面所进行的动作错综复杂，看上去 Python 是要想方设法地进行比较。这种不到黄河不死心的寻求比较的行为会影响比较操作的效率，使得慢速通道与针对整数的快速通道相比，效率上有很大的差别。在我个人看来，实在有点过设计的味道，可能这也是历史遗留的问题。我们就在这里止步吧，不再深入 Pyton 的对象比较体系了，有兴趣的朋友可以深入追下去。

#### 10.1.2.2 比较操作的结果——Python 中的 bool 对象

我们可以略过 Python 的对象比较体系，但是对于比较操作的返回结果需要着重研究一下。在其他编程语言中，比较操作的结果通常会是一个 bool 值，即使在没有内建 bool 值的 C 中，也会使用 1 和 0 来代替 bool 值。Python 虚拟机中也有这样两个对立而统一的代表成功和失败的对象：Py_True 和 Py_False。注意，我说这两个东西是对象，没错，这两个也是名副其实的 PyObject 对象。

```
[boolobject.h]
/* Don't use these directly */
PyIntObject _Py_ZeroStruct, _Py_TrueStruct;

/* Use these macros */
#define Py_False ((PyObject *) &_Py_ZeroStruct)
#define Py_True ((PyObject *) &_Py_TrueStruct)
```

和 C 语言所采用的策略类似，Python 也是利用两个 PyIntObject 对象来充当 bool 对象。

```
[boolobject.c]
/* The type object for bool.  Note that this cannot be subclassed! */
PyTypeObject PyBool_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "bool",
    sizeof(PyIntObject),
    ……
};
/* The objects representing bool values False and True */
/* Named Zero for link-level compatibility */
PyIntObject _Py_ZeroStruct = {
    PyObject_HEAD_INIT(&PyBool_Type)
    0
};

PyIntObject _Py_TrueStruct = {
    PyObject_HEAD_INIT(&PyBool_Type)
    1
};
```

了解了 Python 中比较操作的返回值之后，我们通过图 10-2 看一下从 if a > 10 开始执行，到 COMPRE_OP 指令完成时这一段时间内运行时栈的变化情况。因为在后面，我们马上就会需要利用比较操作的结果来进行指令的跳跃了，那又将是另一个有趣的话题。

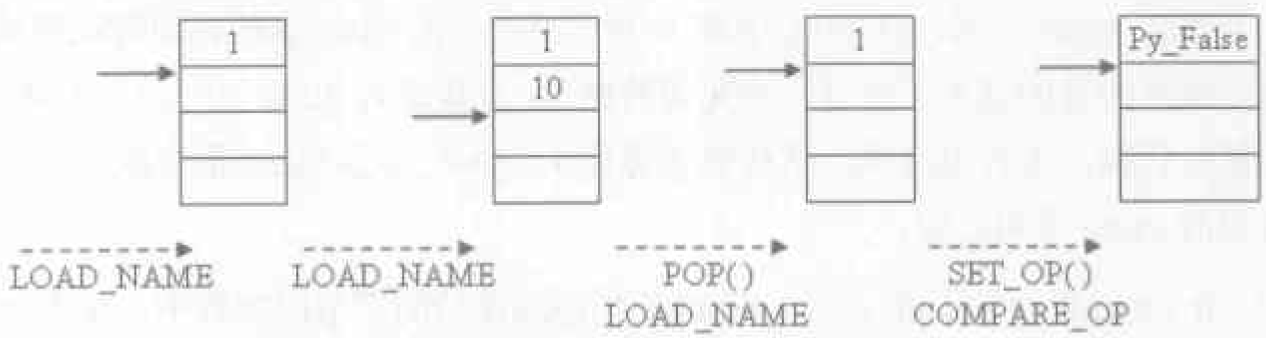



图 10-2 比较操作过程中运行时栈的变化

### 10.1.3 指令跳跃

在 if_control.py 中，如果第一个判断 a > 10 成立，那么会执行接下来的 print 语句；如果判断不成立，则应该跳过 print 语句，执行下一个判断 a <= -2。所以这里有一个指令跳跃的动作。Python 虚拟机中的字节码指令跳跃是如何实现的呢，奥秘就在 COMPARE_OP 指令的实现中最后的那个 PREDICT 宏。

```
[ceval.c]
#define PREDICT(op)     if (*next_instr == op) goto PRED_##op

#define PREDICTED(op)       PRED_##op: next_instr++
```

```
#define PREDICTED_WITH_ARG(op)   PRED_##op: oparg = PEEKARG(); next_instr += 3
#define PEEKARG()  ((next_instr[2]<<8) + next_instr[1])
```

在 Python 中，有一些字节码指令通常都是成对出现的，更准确地说是顺序出现的，这就为根据上一个字节码指令直接预测下一个字节码指令提供了可能。比如 COMPARE_OP 的后面就通常会紧跟 JUMP_IF_FALSE 或 JUMP_IF_TRUE，这一点在 if_control.py 中可以很清晰地看到。而且通常在它们的后面，还会紧跟着一个 POP_TOP 指令。

因此 Python 虚拟机中提供了这样的字节码指令预测功能，如果预测成功，那么会省去很多无谓的操作，使执行效率获得提升。尤其是当这种字节码指令之间的搭配关系出现的概率非常高时，效率的提升尤其显著。

我们可以看到，PREDICT(JUMP_IF_FALSE)，实际就是直接检查下一条待处理的字节码是否是 JUMP_IF_FALSE。如果是，则程序流程会跳转到 PRED_JUMP_IF_FALSE 标识符对应的代码处。将 COMPARE_OP 的实现中的 PREDICT 宏展开，我们可以将这个过程看的更清楚：

```
[COMPARE_OP]
        ......
        if(*next_instr == JUMP_IF_FALSE)
            goto PRED_JUMP_IF_FALSE;
        if(*next_instr == JUMP_IF_FALSE)
            goto PRED_JUMP_IF_TRUE;
```

那么 PRED_JUMP_IF_FALSE 和 PRED_JUMP_IF_TRUE 这些标识符在何处呢？我们知道指令跳跃的目的是为了绕过一些无谓的操作，直接进入 JUMP_IF_FALSE 或 JUMP_IF_TRUE 的指令代码，那么很显然，这些宏应该位于 JUMP_IF_FALSE 指令或 JUMP_IF_TRUE 指令对应的 case 语句之前。

if_control.py 中，在 if a > 10 这条判断编译后的字节码序列中，存在 JUMP_IF_FALSE 指令，那么在 COMPARE_OP 指令的实现代码的最后，将执行 goto PRED_JUMP_IF_ FALSE。所以在这里，我们就来看看 PRED_JUMP_IF_FALSE 标识符是如何被放置到 JUMP_IF_FALSE 指令代码之前的（见代码清单 10-2）。

**代码清单 10-2**

```
[ceval.c]
    PREDICTED_WITH_ARG(JUMP_IF_FALSE);
    case JUMP_IF_FALSE:
        //[1]: 取出之前的比较操作的结果
        w = TOP();
        //[2]: 比较操作结果为 true
        if (w == Py_True) {
            PREDICT(POP_TOP);
            goto fast_next_opcode;
        }
```

```
    //[3]: 比较操作结果为 false，进行指令跳跃
    if (w == Py_False) {
        JUMPBY(oparg);
        goto fast_next_opcode;
    }
    err = PyObject_IsTrue(w);
    if (err > 0)
        err = 0;
    else if (err == 0)
        JUMPBY(oparg);
    else
        break;
    continue;
```

看看那个 PREDICTED_WITH_ARG 宏，我们来把它展开：

```
[ceval.c]
  PRED_JUMP_IF_FALSE:
     //取指令的参数
     oparg = ((next_instr[2]<<8) + next_instr[1]);
     //调整 next_instr
     next_instr += 3;
     case JUMP_IF_FALSE:
       ………
```

当程序的执行流程跳转到 PRED_JUMP_IF_FALSE 处时，首先会通过 PEEKARG() 从字节码指令序列 code 中取出 JUMP_IF_FALSE 指令的指令参数，这个指令参数指示了当某个条件满足时 Python 虚拟机会向前跳跃的字节码指令数。这一点在后面会清晰地展示出来。现在我们只需要记住，这里，我们取了一个指令参数。

值得注意的是，在 PEEKARG 之后，Python 将字节码指针向前移动了 3 个字节的长度。仔细想一下，在 COMPARE_OP 指令的实现中，PREDICT (JUMP_IF_FALSE) 处只是判断下一条字节码是否是 JUMP_IF_FALSE，并没有移动 next_instr。而在接下来的 PEEKARG 中，我们获得了 JUMP_IF_FALSE 的指令参数，也没有移动 next_instr，所以这时当确认应该执行 JUMP_IF_FALSE 时，我们必须将字节码指针移动到 JUMP_IF_FALSE 之后的下一条字节码，因为这时我们已经开始处理 JUMP_IF_FALSE 了，而 next_instr 的使命是指出下一条指令是什么。一个字节码长度为 1 个字节，而字节码的参数都是 2 个字节，所以这里需要将 next_instr 向前移动 3 个字节。

在执行 JUMP_IF_FALSE 指令时，首先会在代码清单 10-2 的[1]处取出运行时栈中栈顶的对象，根据前面的分析，这时是一个 Py_False。然后，就会根据这个对象进行不同的动作。

因为我们现在执行的是 JUMP_IF_FALSE 指令，换句话说，就是“如果为 false，则跳跃”指令。所以如果代码清单 10-2 的[1]处得到的 w 是一个 Py_False，那么很好，匹配上了，就进行跳跃的动作：

```
#define JUMPBY(x)   (next_instr += (x))
```

跳跃的距离就是 JUMP_IF_FALSE 的指令参数，在这里是 9。在第一节所展示出的字节码指令序列中，JUMP_IF_FALSE 那一行的第 4 列形象地给出了该条指令的结果是“to 27”。我们查看一下所有字节码指令所在行的第 1 列，发现第 1 列为 27 的指令是“27 POP_TOP”指令。

现在有个问题，这个 9 是如何确定的呢。按照正常的程序流程，当 a > 10 的判断失败后，会进行下一个判断 a <= -2，这个判断的第一条字节码指令应该是“28 LOAD_NAME 0”，与当前字节码指针 next_instr 所指向的“18 POP_TOP”之间的距离为 10（字节码 1 个字节，参数 2 个字节），如果跳跃距离为 9，那么确实就会跳到 print 语句的最后一条字节码指令“27 POP_TOP”上。

仔细想一想，其实这才是正确的行为。因为在 JUMP_IF_FALSE 的开始，是通过 TOP 抽取运行时栈中的对象的，所以现在那个 Py_False 还停留在栈中，在进行新的判断动作之前，需要将这个对象从栈中弹出，打扫干净屋子，才好迎接新的客人☺。

如果 JUMP_IF_FALSE 执行时从栈中得到的对象是一个 Py_True，这实际上意味着判断 a > 10 是成立的，所以接下来就会执行 print 语句。再次利用 PREDICT 宏，对 POP_TOP 指令进行预测，快速进行运行时栈的清理工作，并为 print 的执行做好准备。

```
[ceval.c]
        PREDICTED(POP_TOP);
        case POP_TOP:
            v = POP();
            Py_DECREF(v);
            goto fast_next_opcode;
```

在 JUMP_IF_FALSE 的指令代码之前，有一个 PREDICTED_WITH_ARG 宏，而在 POP_TOP 的指令代码之前，出现的却是一个 PREDICTED 宏。正如它们的名字所显示出来的区别，这两个宏分别处理有参指令和无参指令两种情况。PREDICTED_WITH_ARG 宏处理有参指令，一定会放置在有参指令之前，比如 JUMP_IF_FALSE；而 PREDICTED 宏处理无参指令，POP_TOP 正是一条无参指令。PREDICTED 完成的工作其实和 PREDICTED_WITH_ARG 并无二致，它的终极目的也是调整 next_instr 的值，使其指向下一条待执行的字节码指令。

值得注意的是，在执行 JUMP_IF_FALSE 时，无论 COMPARE_OP 的结果是 Py_False 还是 Py_True，都将转到 fast_next_opcode，执行下一条字节码指令，而不会接着执行 COMPARE_OP 指令代码中 PREDICT(JUMP_IF_FALSE)其后的 PREDICT（JUMP_IF_ TRUE）。

在整个指令跳跃的过程中，出现了两次跳跃，可能会令人比较迷惑，实际上这两次跳跃是在不同的层面上的跳跃。第一次是通过 PREDICT(JUMP_IF_FALSE)中的 goto 语句进行跳跃，这次跳跃影响的是 Python 虚拟机自身，即实现 Python 的 C 代码。而在 JUMP_IF_FALSE 的指令代码中通过 JUMPBY 完成的跳跃是在 Python 应用程序层面的跳跃，

影响的 Python 应用程序，是.py 源文件中的 Python 代码。

现在我们通过对 PREDICT 宏的分析可以知道，只有如下的字节码指令序列才能使 COMPARE_OP 中的 PREDICT（JUMP_IF_TRUE）得到执行：

```
COMPARE_OP *
JUMP_IF_TRUE *
POP_TOP
```

很遗憾，在 if_control.py 中的所有 if 控制语句都没有编译出这样的指令结构。只有将诸如 if a > 10 这样的判断改为 if not a > 10 时，Python 编译器才会为 if 控制语句编译出含有 JUMP_IF_TRUE 的指令序列。

最后还有一点需要指出的是，在 print 的执行中，同样会有指令跳跃的动作出现。因为从程序的流程上看，在执行完了 print 之后，程序就会跳过以下的几个判断直接执行这个 if 控制结构之后的下一条字节码，所以在执行完 print 之后，Python 虚拟机的执行流程会横越千里，直接跳转到 if 控制结构的末尾后的第一条字节码指令，这个惊险的飞跃由 JUMP_FORWORD 指令完成。

```
[JUMP_FORWARD]
        JUMPBY(oparg);
        goto fast_next_opcode;
```

跳跃的距离是当前字节码与 if 控制结构之后的第一条字节码指令（在 if_control.py 中是倒数第二条指令“99 LOAD_CONST 8”）之间的距离，包括所有字节码指令和其对应的指令参数。所以我们看到，在不同的 print 语句编译后的指令序列中，JUMP_FORWARD 的指令参数是不同的。但是它们跳跃的目标却都是一致的，都是“99 LOAD_CONST 8”。

从这里也可以看到，在 JUMP_FORWARD 后面的那条 POP_TOP 对于 print 语句没有任何意义，它实际上就是为了 JUMP_IF_FALSE 指令而生的。

## 10.2 Python 虚拟机中的 for 循环控制流

在 if 控制结构中，只存在着分支结构，这意味着在 if 控制结构中，只存在字节码指令的前向跳跃。虽然程序有可能左右摇摆，但是程序的流程始终是向前奔流而去。在本节中，我们将研究另一种控制结构：for 循环控制结构。在循环控制结构中，我们将看见一种新的指令跳跃方式，即指令的回退。在上一节中，我们看到了指令跳跃时，通常跳跃的距离都是当前指令与目标指令之间的距离。如果按照这种逻辑，那么在回退时，这个跳跃的距离一定就是负数了，是否真是如此呢？别急，马上就会见分晓了。在进入对 for 循环控制结构的剖析之前，我们先来看一看这一节剖析的对象。

## 10.2.1 研究对象——for_control.py

在这一节，我们通过对 for_control.py 的剖析来研究 Python 中 `for` 循环控制结构的实现。下面列出了包含了字节码指令序列的 for_control.py：

```
[for_control.py]
lst = [1, 2]
0   LOAD_CONST    0 (1)
3   LOAD_CONST    1 (2)
6   BUILD_LIST    2
9   STORE_NAME    0 (lst)
for i in lst:
12   SETUP_LOOP    19 (to 34)
15   LOAD_NAME     0  (lst)
18   GET_ITER
19   FOR_ITER      11 (to 33)
22   STORE_NAME    1 (i)
   print i
25   LOAD_NAME     1 (i)
28   PRINT_ITEM
29   PRINT_NEWLINE
30   JUMP_ABSOLUTE    19
33   POP_BLOCK
34   LOAD_CONST    2 (None)
37   RETURN_VALUE
```

图 10-3 展示了 for_control.py 编译得到的 `PyCodeObject` 中的常象表 `co_consts` 和符号表 `co_names`：


```
- <consts>
    <int value="1" />
    <int value="2" />
    <NoneObject />
  </consts>
- <names>
    <internStr index="0" length="3" value="lst" />
    <internStr index="1" length="1" value="i" />
  </names>
```


图 10-3 for_control.pyc 中的常量表和符号表

## 10.2.2 循环控制结构的初始化

对于 for_control.py 中的第一条 Python 语句，我们已经很熟悉了，不用再去考虑。Python 虚拟机中对 `for` 循环控制流的实现的关键从第二条 Python 语句开始。第一条字节码指令“`12 SETUP_LOOP 19`”是一条关键的指令，正是它拉开了 Python 虚拟机中 `for` 循环控制结构的大幕：

```
[SETUP_LOOP]
PyFrame_BlockSetup(f, opcode, INSTR_OFFSET() + oparg,STACK_LEVEL());
```

#### 10.2.2.1 PyTryBlock

在 SETUP_LOOP 的实现代码中，仅仅是简单地调用了一个 PyFrame_BlockSetup 函数，看来，这个函数的作用至关重要。

```
[frameobject.c]
void PyFrame_BlockSetup(PyFrameObject *f, int type, int handler, int level)
{
    PyTryBlock *b;
    b = &f->f_blockstack[f->f_iblock++];
    b->b_type = type;
    b->b_level = level;
    b->b_handler = handler;
}
```

在这里，我们第一次开始使用 PyFrameObject 中的 f_blockstack：

```
[frameobject.h]
typedef struct _frame {
    ......
    int f_iblock;        /* index in f_blockstack */
    PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks */
    ......
} PyFrameObject;
```

其中的 CO_MAXBLOCKS 在 Python 2.5 中被定义为 20，f_iblock 在调用 PyFrame_New 时被初始化为 0。而那个至关重要的 PyTryBlock 定义如下：

```
[frameobject.h]
typedef struct {
    int b_type;         /* what kind of block this is */
    int b_handler;      /* where to jump to find handler */
    int b_level;        /* value stack level to pop to */
} PyTryBlock;
```

显然，PyFrameObject 对象中的 f_blockstack 是一个 PyTryBlock 结构的数组，而 SETUP_LOOP 指令所做的就是从这个数组中获得了一块 PyTryBlock 结构，并在这个结构中存放了一些 Python 虚拟机当前的状态信息。比如当前执行的字节码指令，当前运行时栈的深度等等。那么这个结构在 for 循环控制结构中起着什么样的作用呢？到目前为止，一无所知。不过，随着对 for_control.py 剖析的深入，我们马上就能看到 PyTryBlock 结构的作用了。

我们注意到 PyTryBlock 结构中有一个 b_type 域，这意味着实际上存在着几种不同用途的 PyTryBlock 对象。从 PyFrame_BlockSetup 中可以看到，这个 b_type 实际上被设置为当前 Python 虚拟机正在执行的字节码指令，以字节码指令作为区分 PyTryBlock 的不同用途。我们看一看有哪些指令会调用 PyFrame_BlockSetup，就能知道 PyTryBlock 有多少种用途了。

```
[ceval.c]
case SETUP_LOOP:
```

```
case SETUP_EXCEPT:
case SETUP_FINALLY:
    PyFrame_BlockSetup(f, opcode, INSTR_OFFSET() + oparg,STACK_LEVEL());
```

原来除了循环控制流，这个小小的 PyTryBlock 在异常机制中也占有一席之地，这里对异常机制我们暂且按下不表，单看 PyTryBlock 在循环控制流中的作用。

### 10.2.2.2 list 的迭代器

在 SETUP_LOOP 指令从 PyFrameObject 的 f_blockstack 中申请了一块 PyTryBlock 结构的空间之后，Python 虚拟机通过"15 LOAD_NAME 0"指令，将刚创建的 PyListObject 对象压入运行时栈。然后再通过执行"18 GET_ITER"指令来获得 PyListObject 对象的迭代器。

```
[GET_ITER]
            //从运行时栈获得 PyListObject 对象
            v = TOP();
            //获得 PyListObject 对象的 iterator
            x = PyObject_GetIter(v);
            Py_DECREF(v);
            if (x != NULL) {
                //将 PyListObject 对象的 iterator 压入堆栈
                SET_TOP(x);
                PREDICT(FOR_ITER);
                continue;
            }
            STACKADJ(-1);
```

在 GET_ITER 的指令代码中，Python 虚拟机首先会获得位于运行时栈的栈顶的 PyListObject 对象，然后通过 PyObject_GetIter 获得 PyListObject 对象的迭代器：

```
[object.h]
typedef PyObject *(*getiterfunc) (PyObject *);

[abstract.c]
PyObject* PyObject_GetIter(PyObject *o)
{
    PyTypeObject *t = o->ob_type;
    getiterfunc f = NULL;
    if (PyType_HasFeature(t, Py_TPFLAGS_HAVE_ITER))
        //获得类型对象中的 tp_iter 操作
        f = t->tp_iter;
    if (f == NULL) {
        ......
    }
    else {
        //通过 tp_iter 操作获得 iterator
        PyObject *res = (*f)(o);
        ......
        return res;
    }
}
```

显然，PyObject_GetIter 是通过调用对象对应的类型对象中的 tp_iter 操作来获得与对象关联的迭代器的。在 Python 中，迭代器的概念实际上与 Java、C#或 C++中 STL 各容器的迭代器的概念是一致的，都是为容器中元素的遍历操作提供一层接口，将对容器的遍历操作与容器的具体实现分离开。这也是 GoF 中的 Iterator Pattern 思想的实现。在 C++ 的 STL 中，有的迭代器是直接利用 C 的原生指针实现的，比如 vector；有的则是通过一个类来实现的，比如 list。虽然 Python 中的 PyListObject 对象在内存布局上与 vector 相同，但其对应的迭代器却和 STL 中的 list 是一致的，都通过额外的对象来实现。实际上，在 Python 中，不光是 PyListObject 对象，只要拥有迭代器的对象，这些迭代器都是一个实实在在的对象，很显然，它们也都是一个 PyObject 对象。

```
[listobject.c]
typedef struct {
    PyObject_HEAD
    long it_index;
    PyListObject *it_seq; /* Set to NULL when iterator is exhausted */
} listiterobject;
```

既然迭代器是一个 PyObject 对象，那么很显然，它也一定拥有类型对象。迭代器 listiterobject 对象所对应的类型对象为 PyListIter_Type。

```
[listobject.c]
PyTypeObject PyListIter_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,                              /* ob_size */
    "listiterator",                 /* tp_name */
    sizeof(listiterobject),         /* tp_basicsize */
    0,                              /* tp_itemsize */
    /* methods */
    ……
    PyObject_SelfIter,                      /* tp_iter */
    (iternextfunc)listiter_next,            /* tp_iternext */
    0,                                      /* tp_methods */
    ……
};
```

我们回到 PyListObject 来，在 PyListObject 对象的类型对象 PyList_Type 中，tp_iter 域被设置为 list_iter。这个 list_iter 正是 PyObject_GetIter 中所获得的那个 f，也正是创建迭代器对象的命门所在：

```
[listobject.c]
static PyObject * list_iter(PyObject *seq)
{
    listiterobject *it;
    it = PyObject_GC_New(listiterobject, &PyListIter_Type);
    it->it_index = 0;
    Py_INCREF(seq);
    //这里 seq 就是之前创建的 PyListObject 对象
    it->it_seq = (PyListObject *)seq;
    _PyObject_GC_TRACK(it);
    return (PyObject *)it;
}
```

PyListObject 对象的迭代器对象只是对 PyListObject 对象做了一个简单的包装，在迭代器中，维护了当前访问的元素在 PyListObject 对象中的序号：it_index。通过这个序号，listiterobject 对象就可以实现对 PyListObject 的遍历。

GET_ITER 指令在获得了 PyListObject 对象的迭代器之后，通过 SET_TOP 宏强制将这个迭代器对象设置为运行时栈的栈顶元素。图 10-4 展现了这个过程：

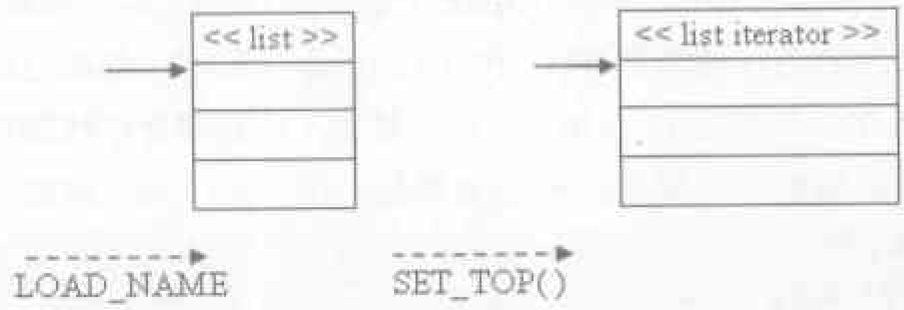



**图 10-4 创建迭代器时运行时栈的变化**

在指令"18 GET_ITER"完成之后，Python 虚拟机开始了 FOR_ITER 指令的预测动作，如你所知，这样的预测动作是为了提高执行的效率。

## 10.2.3 迭代控制

想象一下，对于源代码一级的循环控制结构，在 Python 虚拟机一级，一定也对应着一个相应的循环控制结构。因为无论进行怎样的变换，都不可能在虚拟机一级利用顺序结构来实现源码一级的循环结构，这可以看做程序的拓扑不变性。到现在为止，我们看到的都是顺序结构，很显然，创建迭代器的指令应该也属于顺序结构中的指令，因为如果每一次循环都要创建迭代器，那效率跟乌龟没什么两样了。尽管"乌龟"在"龟兔赛跑"中大获全胜，但如果去跟 Ruby、Perl 或者 PHP 赛跑，恐怕就没那么幸运了。

而另一方面，从概念上来说，迭代器正是一个对容器实现遍历的工具。遍历，就是循环的另一种说法。我们现在可以猜想，在 GET_ITER 指令之后，Python 虚拟机应该进入一个与源码对应的循环控制结构了。没错，正是从"19 FOR_ITER 11"指令开始，我们开始进入 Python 虚拟机一级的 for 循环：

```
[FOR_ITER]
        PREDICTED_WITH_ARG(FOR_ITER);
        case FOR_ITER:
            //从运行时栈的栈顶获得 iterator 对象
            v = TOP();
            //通过 iterator 对象获得集合中的下一个元素对象
            x = (*v->ob_type->tp_iternext)(v);
            if (x != NULL) {
                //将获得的元素对象压入运行时栈
                PUSH(x);
                PREDICT(STORE_FAST);
```

```
            PREDICT(UNPACK_SEQUENCE);
            continue;
        }

        /* iterator ended normally */
        //x == NULL，意味着 iterator 的迭代已经结束
        x = v = POP();
        Py_DECREF(v);
        JUMPBY(oparg);
        continue;
```

FOR_ITER 的指令代码会首先从运行时栈中获得 PyListObject 对象的迭代器，然后调用迭代器的 tp_iternext 开始进行迭代。迭代器的 tp_iternext 操作总是返回与迭代器关联的容器对象中的下一个元素，如果当前已经抵达了容器对象的结束位置，那么 tp_iternext 将返回 NULL，这个结果预示着遍历结束。

FOR_ITER 的指令代码会检查 tp_iternext 的返回结果，如果得到的是一个有效的元素（x != NULL），那么将获得的这个元素压入到运行时栈中，并开始进行一系列的字节码预测动作。在我们的例子中，这两个预测动作显然都会失败，所以在 continue 处，Python 虚拟机会重新进入对下一条字节码指令——“22 STORE_NAME 1”——的执行过程。

对于 PyListObject 对象的迭代器，其获取下一个元素的操作如下：

```
[listobject.c]
static PyObject* listiter_next(listiterobject *it)
{
    PyListObject *seq;
    PyObject *item;
    //注意这里的 seq 是一个 PyListObject 对象
    seq = it->it_seq;

    if (it->it_index < PyList_GET_SIZE(seq)) {
        //获得序号为 it_index 的元素对象
        item = PyList_GET_ITEM(seq, it->it_index);
        //调整 it_index，使其指向下一个元素对象
        ++it->it_index;
        Py_INCREF(item);
        return item;
    }
    //迭代结束
    Py_DECREF(seq);
    it->it_seq = NULL;
    return NULL;
}
```

在获取了当前 it_index 对应的 PyListObject 对象中的元素后，将 it_index 递增，为下一个迭代动作做好准备。在我们的例子中，这里会返回一个 PyIntObject 对象 1。

图 10-5 展示了直到“22 STORE_NAME 1”之前，运行时栈及 local 名字空间的变化情况。

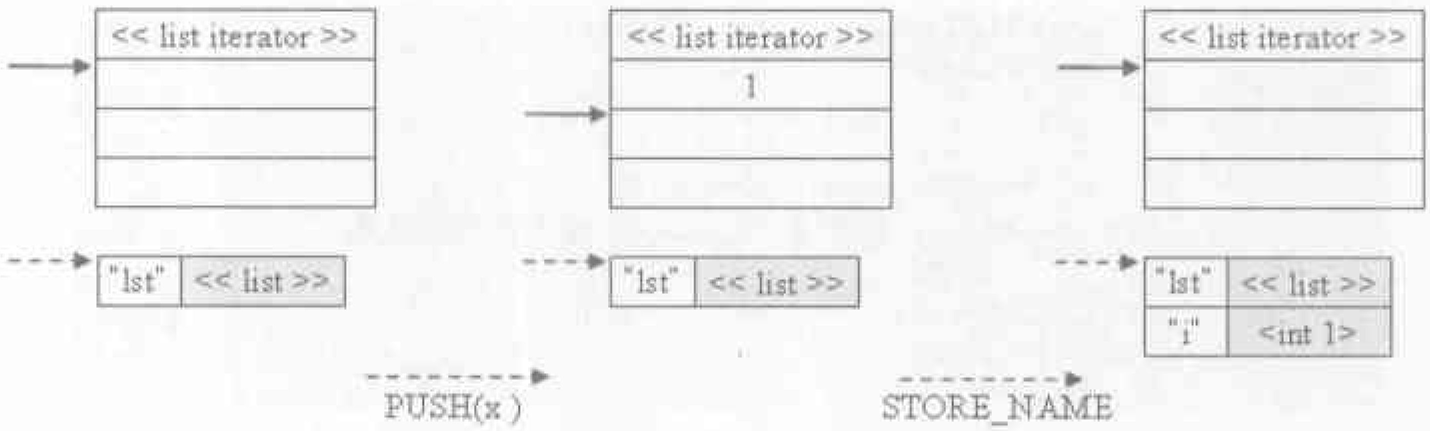



图 10-5 迭代过程中虚拟机的状态变化

这之后 Python 虚拟机将沿着字节码的顺序一条一条执行下去，从而完成输出的动作。然后，按照 for_control.py 中 Python 源代码所定义的行为，应该获得 PyListObject 对象中的下一个元素，并继续进行输出操作。直观上，这时需要一个向后回退的指令，这个指令正是"30 JUMP_ABSOLUTE 19"：

```
[JUMP_ABSOLUTE]
JUMPTO(oparg);

#define JUMPTO(x)   (next_instr = first_instr + (x))
```

前面我们提到，如果 for 循环中的指令跳跃动作依然采用 JUMPBY 的逻辑，那么参数必须是一个负数，因为这里的指令跳跃式一个向后回退的过程。但是 Python 采用了另外一种做法，不再使用相对距离跳跃，而是使用了基于字节码指令序列开始位置的绝对距离跳跃。从名字就可以看出，完成这个绝对跳跃动作的指令正是 JUMP_ABSOLUTE 指令。JUMP_ABSOLUTE 指令的行为是强制设定 next_instr 的值，将 next_instr 设定到距离 f->f_code->co_code 开始地址的某一特定偏移的位置。这个偏移的量由 JUMP_ABSOLUTE 的指令参数决定，所以，这条参数就成了 for 循环中指令回退动作最关键的一点。在 for_control.py 中，参数的值是 19，那么这个 19 代表着什么呢？

我们从字节码指令序列的开始处（f->f_code->co_code），向前前进 19 个字节，其中包括字节码和参数，19 个字节之后，到达了"19 FOR_ITER 11"。也就是说，在 Python 虚拟机执行完"JUMP_ABSOLUTE 19"之后，next_instr 将指向"19 FOR_ITER 11"这条指令，那 Python 虚拟机的下一步动作就是执行 FOR_ITER 指令，即通过 PyListObject 对象的迭代器获得 PyListObject 对象中的下一个元素，然后依次向前，继而执行输出操作。

可见，在 JUMP_ABSOLUTE 指令处，Python 虚拟机实现了字节码的向后回退动作。而 Python 虚拟机也在 FOR_ITER 指令和 JUMP_ABSOLUTE 指令之间成功地构造出了一个循环结构，正是这个循环结构对应着 for_control.py 中的那个 for 循环结构。

细心的读者可能已经注意到了一个奇怪的现象，在"19 FOR_ITER 11"这条指令中，

很明显，FOR_ITER 是一个带参数的指令，但是到目前为止，我们都没有看到这个参数有什么用。列位看官，别急，关于这个参数的作用，马上就会水落石出了。

## 10.2.4 终止迭代

俗话说“天下无不散的宴席”。在 control.py 中，for 循环控制结构最终也是要结束的，这个循环结束的行为同样是要落在 FOR_ITER 的身上。在 FOR_ITER 指令的执行过程中，如果通过 PyListObject 对象的迭代器获得的下一个元素不是有效的元素（NULL），这就意味着迭代结束了。这个结果将直接导致 Python 虚拟机会将迭代器对象从运行时栈中弹出，同时执行一个 JUMPBY 的动作，向前跳跃。

```
#define JUMPBY(x)   (next_instr += (x))
```

向前跳跃的距离即是 FOR_ITER 的参数，在 for_control.py 中是 11。我们从 FOR_ITER 后的 STORE_NAME 向前前进 11 个字节，正好到达 POP_BLOCK：

```
[POP_BLOCK]
        {
            //将运行时栈恢复为迭代前的状态
            PyTryBlock *b = PyFrame_BlockPop(f);
            while (STACK_LEVEL() > b->b_level) {
                v = POP();
                Py_DECREF(v);
            }
        }

[frameobject.c]
PyTryBlock *PyFrame_BlockPop(PyFrameObject *f)
{
    PyTryBlock *b;
    //向 f_blockstack 中归还 PyTryBlock
    b = &f->f_blockstack[--f->f_iblock];
    return b;
}
```

有意思的是，POP_BLOCK 的行为与它的名字有点不同。前面我们看到，在 SETUP_LOOP 处，Python 虚拟机从 f->f_blockstack 中申请了一个 PyTryBlock 结构，并将执行 SETUP_LOOP 时的一些关于虚拟机的状态信息保存到了所获得的 PyTryBlock 结构中。在执行 POP_BLOCK 指令时，实际上是将这个 PyTryBlock 结构归还给了 f->f_blockstack。同时，Python 虚拟机抽取在 SETUP_LOOP 指令处保存在 PyTryBlock 中的信息，并根据其中存储的 SETUP_LOOP 指令处运行时栈的深度信息将运行时栈恢复到 SETUP_LOOP 之前的状态，从而完成了整个 for 循环结构。在这之后，就开始执行 for_control.py 中 for 循环之后的那个隐藏的返回动作了。

我们注意到，在执行 SETUP_LOOP 指令时，Python 虚拟机保存了许多信息，而在执行

POP_BLOCK 指令时，却只使用了栈深度信息来恢复运行时栈。为什么会存在这种不对称呢？这是因为，PyTryBlock 并不是专门为 for 循环准备的，Python 中还有一些机制会使用到这个结构，为了避免代码过于复杂，Python 不管三七二十一，在 PyFrame_BlockSetup 中会一古脑将所有机制可能用到的参数全都存放到 PyTryBlock 结构中，各个机制需要什么参数，取用需要的参数即可，对于其他参数，可以不用理会。

最后，我们修改了 Python 的源代码，来实时地观察一下 for 循环的整个执行流程，结果如图 10-6 所示。由于 IDLE 自身使用了多线程，而其他线程处于活动状态时同样会执行 for 循环语句，从而导致我们无法观察到正常的结果，所以这里我们是在命令行模式下进行观察的。在图 10-6 创建的 list 对象中，第一个元素"654321"是我们修改的代码中用来判断是否该输出信息的准则。

```
>>> lst = [654321, 3, 2, 1]
>>> for i in lst:
...     pass
...
[GET_ITER] : Get iterator...
[FOR_ITER] : Get next item -> 654321...
[JUMP_ABSOLUTE] : Go back to FOR_ITER...
[FOR_ITER] : Get next item -> 3...
[JUMP_ABSOLUTE] : Go back to FOR_ITER...
[FOR_ITER] : Get next item -> 2...
[JUMP_ABSOLUTE] : Go back to FOR_ITER...
[FOR_ITER] : Get next item -> 1...
[JUMP_ABSOLUTE] : Go back to FOR_ITER...
[FOR_ITER] : Get next item -> NULL...
>>>
```

图 10-6 监视 Python 虚拟机执行 for 循环的过程

## 10.3 Python 虚拟机中的 while 循环控制结构

在大多数现代程序设计语言中，一般都提供了四种最基本的控制结构：if 条件控制结构、for 循环结构、while 循环结构和 swich 分支选择控制结构。虽然 Python 开发社区也曾有过关于在 Python 中实现 swich 机制的讨论，但是到目前最新的 Python 发行版本，都不存在 switch 控制结构。所以，我们只剩下最后一种基本的控制结构了，即 while 循环控制结构。

### 10.3.1 研究对象——while_control.py

前面我们已经完成了对 Python 中 if 条件控制结构和 for 循环结构的考察，那么现在

对于 Python 中的 `while` 循环控制结构，其实已经很好理解了。在这一节，我们通过对 while_control.py 来考察 Python 中的 `while` 循环控制结构。在 while_control.py 中，我们不仅要考虑循环本身的指令跳跃动作，而且还要考虑另外两个与循环相关的指令跳跃语义：`continue` 和 `break`。

下面是经编译生成了字节码的 while_control.py：

```
[while_control.py]
i = 0
0  LOAD_CONST   0 (0)
3  STORE_NAME   0 (i)
while i < 10:
6   SETUP_LOOP   71 (to 80)
9   LOAD_NAME    0 (i)
12  LOAD_CONST   1 (10)
15  COMPARE_OP   0 (<)
18  JUMP_IF_FALSE   57 (to 78)
21  POP_TOP
   i += 1
22  LOAD_NAME    0 (i)
25  LOAD_CONST   2 (1)
28  INPLACE_ADD
29  STORE_NAME   0 (i)
   if a > 5:
32  LOAD_NAME    0 (i)
35  LOAD_CONST   3 (5)
38  COMPARE_OP   2 (==)
41  JUMP_IF_FALSE  7 (to 51)
44  POP_TOP
      continue
45  JUMP_ABSOLUTE 9
48  JUMP_FORWARD 1 (to 52)
51  POP_TOP
   if a == 20:
52  LOAD_NAME    0 (i)
55  LOAD_CONST   4 (20)
58  COMPARE_OP   2 (==)
61  JUMP_IF_FALSE  5 (to 69)
64  POP_TOP
      break
65  BREAK_LOOP
66  JUMP_FORWARD  1 (to 70)
69  POP_TOP
   print i
70  LOAD_NAME    0 (i)
73  PRINT_ITEM
74  PRINT_NEWLINE
75  JUMP_ABSOLUTE   9
78  POP_TOP
79  POP_BLOCK
80  LOAD_CONST   3
83  RETURN_VALUE
```

编译后产生的 PyCodeObject 对象中的常量表 co_consts 和符号表 co_names 如图 10-7 所示：

```
- <consts>
    <int value="0" />
    <int value="10" />
    <int value="1" />
    <int value="5" />
    <int value="20" />
    <NoneObject />
  </consts>
- <names>
    <internStr index="0" length="1" value="i" />
  </names>
```

图 10-7 while_control.pyc 中的常量表和符号表

字节码指令序列的结构基本都是我们已经熟悉的了，在“6 SET_LOOP 71”处，虚拟机从当前活动的 PyFrameObject 对象中申请了一块 PyTryBlock 的空间，并填入一些当前虚拟机的状态，然后正式开始进入 while 循环。所以这里我们只观察 while 循环中运行时栈和 local 名字空间在运行时的变化情况。

## 10.3.2 循环终止

当 Python 虚拟机执行到“15 COMPARE_OP 0”指令时，运行时栈及局部变量表的情况如图 10-8 所示：

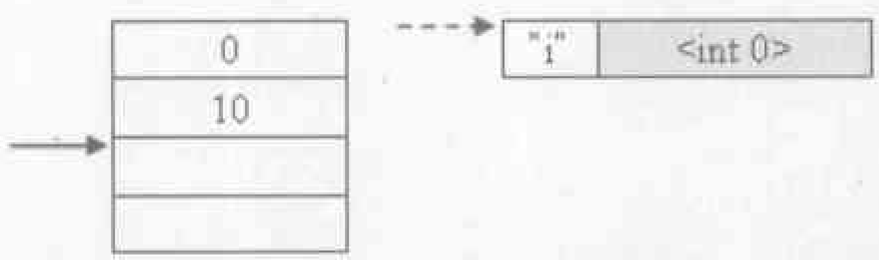



图 10-8 执行“15 COMPARE_OP 0”指令时虚拟机的状态

接着“COMPARE_OP 0”指令将执行“小于”比较操作，并将比较的结果存放到运行时栈中。这里，我们先来个思维的跃迁，直接考虑循环结束时的情况。当某个时刻 i 的值开始大于或等于 10 时，“15 COMPARE_OP 0”指令运行后的运行时栈和 local 名字空间情况如图 10-9 所示：

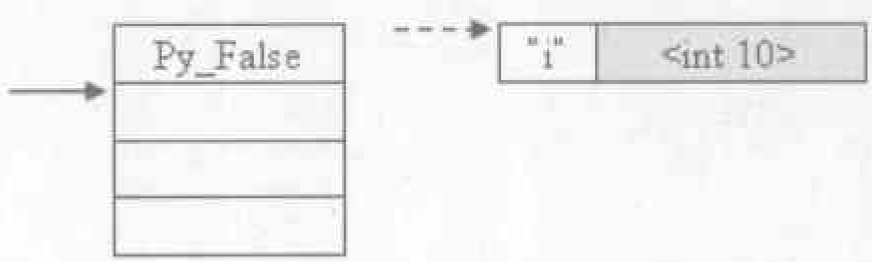



图 10-9 循环结束时虚拟机的状态

虚拟机在执行紧接着的“18 JUMP_IF_FALSE 57”时，会发生指令跳跃的动作，跳跃的距离是 57，从该指令行的第四列可以看到 dis 工具解析的结果，这个距离正好到了倒数第 4 条字节码指令“78 POP_TOP”处，显然这条指令将 Py_False 弹出运行时栈，随后的虚拟机通过执行 POP_BLOCK 指令销毁 PyTryBlock 对象。

## 10.3.3 循环的正常运转

另一方面，如果“15 COMPARE_OP 0”指令处的判断结果为 Py_True，那么虚拟机将进入 while 循环。这里我们不考虑 continue 和 break 的情况（实际上，break 永远也不会发生），只考虑循环正常运转时的情况。在循环正常运转时，i 的值小于 10，且不等于 5，那么 Python 虚拟机接下来的动作及状态转换如图 10-10 所示：

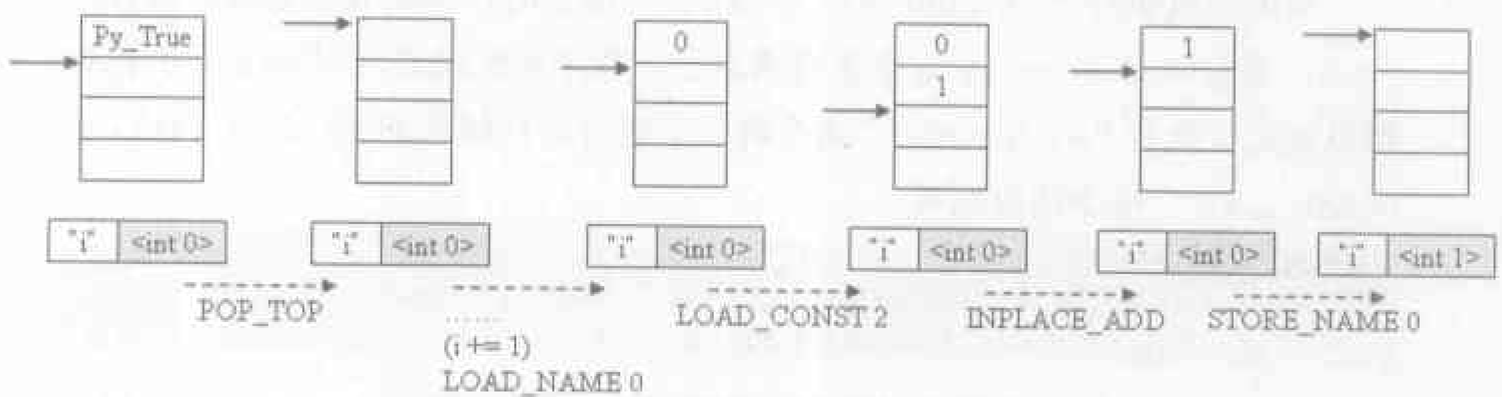



图 10-10 循环运转过程中 Python 虚拟机的状态转换

将 i 的值递增之后，Python 虚拟机通过两个 PRINT_*指令将 i 值输出到标准输出，然后通过执行指令“75 JUMP_ABSOLUTE 9”进行指令的回退，回退的位置是距离字节码开始处 9 个字节，正好是“while i < 10:”下面的“9 LOAD_NAME 0”指令。Python 虚拟机又开始了新一轮的循环，继续比较 i 和 10 的大小，如此反复，在每一次 TRUE 分支中，都会使 i 的值递增 1，直到 i 的值等于 10。这时程序的执行流程就会转入 FALSE 分支退出循环，然后 Python 的虚拟机才会执行 while 循环控制结构之后的字节码指令序列。

## 10.3.4 循环流程改变指令之 continue

在 while_control.py 中，我们设置了一个执行 continue 动作的条件，即 i == 5。当 Python 虚拟机执行 continue 动作时，按照在其他编程语言，比如 C、Java 中的经验，这时程序的执行流程应该跳转到循环开始的地方，而不接着执行循环中位于 continue 之后的语句。

在指令序列中，我们可以看到，当 i == 5 时，也就意味着“41 JUMP_IF_FALSE 7”指令中的判断操作的结果为 Py_True，那么这里的指令跳跃动作将不会发生，所以虚拟机将执行接下来的“44 POP_TOP”和“45 JUMP_ABSOLUTE 9”指令。在执行 JUMP_ABSOLUTE

指令时，虚拟机的流程就跳转到了“9 LOAD_NAME 0”指令处，这完全符合其他语言中 continue 的语义。

## 10.3.5 循环流程改变指令之 break

虽然我们在 while_control.py 中也设置了另一条能够改变循环流程的语句——break 语句。但是很显然，这条语句永远也不会执行。因为这条语句执行的前提条件是 i == 20，但是当 i == 10 时，while 循环就已经结束了。

尽管 break 语句永远不能执行，但我们从理论上看一看它执行时的动作也是可以的。按照其他编程语言中的 break 语义，执行这条语句将跳出最近的一层循环。

当虚拟机执行“61 JUMP_IF_FALSE 5”指令时，如果其中的判断操作的结果为 Py_True，就意味着 i == 20 这个条件满足了，程序流程就进入对 break 的执行。Python 虚拟机首先会执行“64 POP_TOP”指令将位于运行时栈栈顶的 Py_True 弹出，然后执行“65 BREAK_LOOP”指令结束循环。

```
[BREAK_LOOP]
            why = WHY_BREAK;
            goto fast_block_end;
```

Python 虚拟机将我们之前提到的结束状态 why 设置为 WHY_BREAK，然后进入 fast_block_end 标志符对应的代码处。这处代码是一段比较复杂的代码，其中还有关于异常机制的代码，这里我们只截取与 break 相关的代码：

```
[fast_block_end in PyEval_EvalFrameEx]
fast_block_end:
        while (why != WHY_NOT && f->f_iblock > 0) {
            //取得与当前 while 循环对应的 PyTryBlock
            PyTryBlock *b = PyFrame_BlockPop(f);
            ......
            //将运行时栈恢复到 while 循环之前的状态
            while (STACK_LEVEL() > b->b_level) {
                v = POP();
                Py_XDECREF(v);
            }
            //处理 break 语义动作
            if (b->b_type == SETUP_LOOP && why == WHY_BREAK) {
                why = WHY_NOT;
                JUMPTO(b->b_handler);
                break;
            }
            ......
        }

#define JUMPTO(x)  (next_instr = first_instr + (x))
```

Python 虚拟机首先获得之前通过 SETUP_LOOP 指令申请得到的，与当前 while 循环对应的 PyTryBlock 结构，然后根据其中存储的运行时栈信息将运行时栈恢复到 while 循环之后的状态。最后 Python 虚拟机将 why 设置为 WHY_NOT，表明退出状态没有任何错误，再通过 JUMPTO 宏，将虚拟机种下一条指令的指示器 next_instr 设置为距离 code 开始位置 b->b_handler 个字节的指令。

这个 b_handler 是在执行 SETUP_LOOP 指令时设置的，参考 SETUP_LOOP 的指令代码和 PyFrame_BlockSetup 的代码可以看到，这个值会被设置为 INSTR_OFFSET() + oparg。注意这里的 oparg 是指令“6 SETUP_LOOP 71”的指令参数，即 71。INSTR_OFFSET()宏对应的代码为((int)(next_instr - first_instr))，因为在执行 SETUP_LOOP 指令时，next_instr 已经指向了下一条待执行的字节码指令，即“9 LOAD_NAME 0”，很显然，这里的 b_handler 的值为：INSTR_OFFSET()+oparg = 9 + 71 = 80。这也就意味着 break 语句将导致 Python 虚拟机的执行流程跳转到前面所示的指令序列的倒数第 2 条指令“80 LOAD_CONST 3”处。确实，它已经跳出了 while 循环所对应的指令序列。

值得注意的是，虽然这里使用了 why 这个用于栈帧（PyFrameObject）结束时的结束状态，但是实际上并没有结束当前活动的栈帧，而仅仅是利用其实现了 break 的语义。可以看到，最后 why 又被设置为了正常状态 WHY_NOT，而虚拟机仍然在当前栈帧中运行。

到了这里，我们已经考察完了 Python 中四种基本的程序结构：顺序结构、if 条件分支控制结构、for 循环控制结构和 while 循环控制结构。实现了这四种控制流结构，一个最“轻量级”的脚本语言就可以诞生了。但是在 Python 中，除了这些基本的控制流，还有一种高级的控制流结构，这也是我们将在下面详细考察的内容——异常控制结构。

# 10.4 Python 虚拟机中的异常控制流

程序在运行的过程中，可能经常遇到意想不到的情况，比如说除数为 0、想要打开的文件不存在、正在使用的 socket 非正常关闭等。一个健壮的程序必须处理这些异常的情况。在现代编程语言中，都引入了“异常”这样的概念，来对程序运行中突发的非正常情况进行抽象。同时，在这些编程语言中，还提供了相应的语法结构和语义元素，使得程序员能够通过这些语法结构和语义元素来方便地描述异常发生时程序的行为。

## 10.4.1 Python 中的异常机制

### 10.4.1.1 Python 虚拟机自身抛出异常

Python 内部有一套内建的异常捕捉机制，即使在 Python 脚本文件中没有出现诸如 try、

except、finally 等用于进行异常控制的语义元素，Python 脚本执行中所抛出的异常还是会被 Python 虚拟机捕捉到。我们首先通过一个最简单的例子来考察 Python 中的异常机制是如何实现的：

```
1/0
#   LOAD_CONST   0
#   LOAD_CONST   1
#   BINARY_DIVIDE
```

由于除数是 0，所以这行 Python 代码一定会抛出异常，在 Python 中，这个异常是 ZeroDivideError。我们看一下图 10-11 所示的执行结果：

```
>>> 1/0

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    1/0
ZeroDivisionError: integer division or modulo by zero
```

图 10-11 Python 内建的异常捕捉机制

在这一节，我们就会将异常的来龙去脉搞个一清二楚。在"1/0"这行简单的 Python 表达式编译得到的三条字节码指令中，前两条我们已经非常熟悉了，在第三条字节码指令中，执行了除法操作。显然，异常也正是在执行这条字节码时被触发的：

```
[BINARY_DIVIDE]
            w = POP(); // 1
            v = TOP(); // 0
            x = PyNumber_Divide(v, w);
            Py_DECREF(v);
            Py_DECREF(w);
            SET_TOP(x);
            if (x != NULL) continue;
            break;
```

从运行时栈中获得数据之后，w 是 PyIntObject 对象 1，v 是 PyIntObject 对象 0，x 是做除法操作的结果。我们注意到在将这个结果压入到栈中后，会对 x 的有效性进行检察。如果 x 是一个有效的 Python 对象，那么 Python 虚拟机将执行下一条字节码；如果 x 为 NULL，即不是一个有效的 Python 对象，那么将通过 break 跳出 Python 虚拟机中那个对字节码指令进行分派的巨大的 switch 语句。注意，这里的跳跃最终将使得程序的流程跳出与 Python 虚拟机息息相关的用于指令分派的 switch 选择。

我们可以猜想，当 PyNumber_Divide 执行时，抛出了异常，而它的返回值一定是 NULL，所以才能导致 Python 虚拟机退出当前栈帧。那么这个至关重要的异常在哪里抛出的，它又被抛到了什么地方去了呢？列位看官，且随我深入 PyNumber_Divide。

从 PyNumber_Divide，会经过一系列的动作，在这个过程中，不同的参与除法操作的对象最终将走上不同的路径，而我们的两个 PyIntObject 的除法路径最终会达到在

PyInt_Type 中定义的除法操作 int_classic_div：

```
[intobject.c]
static PyObject* int_classic_div(PyIntObject *x, PyIntObject *y)
{
    long xi, yi;
    long d, m;
    //将 x，y 中维护的整数值转存到 xi，yi 中
    CONVERT_TO_LONG(x, xi);
    CONVERT_TO_LONG(y, yi);
    switch (i_divmod(xi, yi, &d, &m)) {
    case DIVMOD_OK:
        return PyInt_FromLong(d);
    case DIVMOD_OVERFLOW:
        return PyLong_Type.tp_as_number->nb_divide((PyObject *)x,
                            (PyObject *)y);
    default:
        return NULL;
    }
}
```

现在我们可以确定，在 i_divmod 中，抛出了异常，并且返回了一个不是 DIVMOD_OK 或 DIVMOD_OVERFLOW 的值，这样 int_classic_div 才能返回一个表示除法失败的 NULL 值，将 Python 虚拟机扼杀在执行的路上。我们就来看看这个异常发生的 i_divmod：

```
[intobject.c]
/* Return type of i_divmod */
enum divmod_result {
    DIVMOD_OK,      /* Correct result */
    DIVMOD_OVERFLOW,    /* Overflow, try again using longs */
    DIVMOD_ERROR       /* Exception raised */
};

static enum divmod_result
i_divmod(register long x, register long y, long *p_xdivy, long *p_xmody)
{
    long xdivy, xmody;
    //抛出异常的瞬间
    if (y == 0) {
        PyErr_SetString(PyExc_ZeroDivisionError,
                "integer division or modulo by zero");
        return DIVMOD_ERROR;
    }
    ……
}
```

现在，我们终于可以好好看看异常抛出的瞬间。当 i_divmod 发现除数居然是 0 时，调用 PyErr_SetString 抛出了异常，并返回了指示异常抛出的指示码——DIVMOD_ERROR。

乍一看，这个 PyExc_ZeroDivisionError 就是我们苦苦寻找的那个异常。我们说过，Python 中，一切东西都是对象，异常自然也不能例外。那么在 Python 的对象体系中，这个 PyExc_ZeroDivisionError 到底是属于哪一部分的呢？

实际上，这个 PyErr_ZeroDivisionError 很简单，就是一个 PyObject*，仅仅是一个指针：

```
[pyerrors.h]
PyObject *PyExc_ZeroDivisionError;
```

在 pyerrors.h 中，同时还定义了许多在异常机制中使用的 PyObject*：

```
[pyerrors.h]
......
PyObject *PyExc_KeyboardInterrupt;
PyObject *PyExc_MemoryError;
PyObject *PyExc_NameError;
PyObject *PyExc_OverflowError;
PyObject *PyExc_RuntimeError;
PyObject *PyExc_NotImplementedError;
PyObject *PyExc_SyntaxError;
......
```

尽管它们都是再简单不过的 PyObject*，但是在 Python 运行环境初始化时，它们会指向 Python 创建的异常类型对象，从而指明发生了什么异常。

#### 10.4.1.2 在线程状态对象中记录异常信息

在 i_divmod 之后，Python 的执行路径会沿着 PyErr_SetString、PyErr_SetObject，一直到达 PyErr_Restore。在 PyErr_Restore 中，Python 将这个异常放置到了一个安全的地方：

```
[errors.c]
void PyErr_Restore(PyObject *type, PyObject *value, PyObject *traceback)
{
    PyThreadState *tstate = PyThreadState_GET();
    PyObject *oldtype, *oldvalue, *oldtraceback;
    //保存以前的异常信息
    oldtype = tstate->curexc_type;
    oldvalue = tstate->curexc_value;
    oldtraceback = tstate->curexc_traceback;
    //设置当前的异常信息
    tstate->curexc_type = type;
    tstate->curexc_value = value;
    tstate->curexc_traceback = traceback;
    //抛弃以前的异常信息
    Py_XDECREF(oldtype);
    Py_XDECREF(oldvalue);
    Py_XDECREF(oldtraceback);
}

void PyErr_SetObject(PyObject *exception, PyObject *value)
{
    Py_XINCREF(exception);
    Py_XINCREF(value);
    PyErr_Restore(exception, value, (PyObject *)NULL);
```

```
}

void PyErr_SetString(PyObject *exception, const char *string)
{
    PyObject *value = PyString_FromString(string);
    PyErr_SetObject(exception, value);
    Py_XDECREF(value);
}
```

最后，在 PyThreadState 的 curexc_type 中存放下了 PyExc_ZeroDivisionError，而 curexc_value 中存放下了在 i_divmod 中设定的那个跟随 PyExc_ZeroDivisionError 的字符串"integer division or modulo by zero"。

我们在前面已经介绍了 PyThreadState 对象，Python 无论它多么强悍，总会在一个操作系统提供的线程中运行。真实的线程及其状态当然是由操作系统来维护和管理的，但是 Python 虚拟机在运行中总需要另外一些与线程相关的状态和信息，比如是否发生了异常这样的信息。这些信息显然没法由操作系统提供，而 PyThreadState 对象正是 Python 为线程准备的在 Python 虚拟机一级保存线程状态信息的对象。在这里，当前活动线程对应的 PyThreadState 对象可以通过 PyThreadState_GET 获得，在得到了线程状态对象之后，就将异常信息存放到线程状态对象中。

```
[pystate.h]
#define PyThreadState_GET() (_PyThreadState_Current)

[pystate.c]
PyThreadState *_PyThreadState_Current = NULL;
```

在 Python 启动，进行初始化的时候，会调用 PyThreadState_New 创建一个新的 PyThreadState 对象，并将其赋给_PyThreadState_Current，这个对象就是和当前的活动线程关联的线程状态对象。

在 Python 的 sys 标准库中，提供了一个接口，使我们能够在异常发生时，访问 Python 虚拟机存放在线程状态对象中的异常信息。下面的例子展示了这个接口的使用：

```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> try:
...     1/0
... except Exception:
...     import sys
...     print sys.exc_info()[0] //获得 tstate->curexc_type
...     print sys.exc_info()[1] //获得 tstate->curexc_value
...
<type 'exceptions.ZeroDivisionError'> //sys.exc_info()[0]的结果
integer division or modulo by zero    //sys.exc_info()[1]的结果
```

### 10.4.1.3 展开栈帧

我们看到异常已经被记录在了线程的状态对象中了。那么现在可以回头看看，在跳出了分派字节码指令的 switch 块之后，发生了什么动作。这里还存在一个问题，导致跳出那个巨大的 switch 块的原因可能是执行完了字节码之后正常的跳出，也可能是发生异常后的跳出。那么 Python 虚拟机将如何区分呢？

```
[ceval.c]
PyObject* PyEval_EvalFrameEx(PyFrameObject *f)
{
  ……
  for (;;) {
        //巨大的switch语句
        if (why == WHY_NOT) {
            if (err == 0 && x != NULL) {
                  continue; //没有异常情况发生，执行下一条字节码指令
            }
            //设置why，通知虚拟机，异常发生了
            why = WHY_EXCEPTION;
            x = Py_None;
            err = 0;
        }
        //尝试捕捉异常
        if (why != WHY_NOT)//[1]
            break;
        ……
    } //end of for(;;)
    ……
}
```

在跳出了 switch 之后，首先会通过检查 x 的值，如果 x 为 NULL，表示有异常情况发生，那么 Python 虚拟机将 why 设置为 WHY_EXCEPTION。这里的 x 是 PyNumber_Divide 的结果，我们刚才看到，在抛出了异常之后，这个 x 就成了 NULL。变量 why 实际上维护的是 Python 虚拟机中执行字节码指令的那个 for 循环内的状态。当为 WHY_NOT 时，表示一切正常，没有错误发生；而设置成了 WHY_EXCEPTION 之后，表示在执行字节码的过程中，有异常被抛出了。注意，到了这里，Python 的虚拟机才开始获得了这个信息，即**执行过程中发生了异常**。

在 Python 虚拟机意识到有异常发生后，它就要开始进入异常处理的流程，这个流程会涉及我们介绍 PyFrameObject 对象时所提到的那个 PyFrameObject 对象链表。在介绍 PyFrameObject 对象时，我们提到，PyFrameObject 对象是 Python 虚拟机中对栈帧的模拟，当发生函数调用时，Python 虚拟机会创建一个与被调用函数对应的 PyFrameObject 对象（栈帧），并通过该对象中的 f_back 连接到调用者对应的 PyFrameObject 对象。这样就形成了一条 PyFrameObject 对象的链表。

前面我们在考察异常时并没有涉及函数调用，现在我们考虑一下如果函数调用时发生

了异常：

```
def h():
    1/0

def g():
    h()

def f():
    g()

f()
```

图 10-12 展示了当 Python 虚拟机执行到函数 h 中的 1/0 表达式时，所形成的 PyFrame-Object 对象链表：

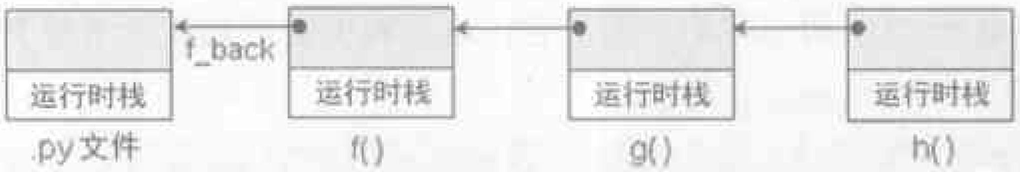



图 10-12 虚拟机执行函数 h 时的栈帧链表

图 10-13 给出了这个脚本运行时产生的输出：

```
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    f()
  File "<pyshell#29>", line 2, in f
    g()
  File "<pyshell#26>", line 2, in g
    h()
  File "<pyshell#23>", line 2, in h
    1/0
ZeroDivisionError: integer division or modulo by zero
```

图 10-13 异常在函数调用中发生

可以看到，在输出的信息中，出现了函数调用的信息：比如在源代码的哪一行调用了函数，调用了什么函数，这些信息是如何得来的呢？同时我们注意到，输出的信息俨然呈现出一种链状的结构，如同图 10-12 中所展示的栈帧链表一样，这两者之间难道有什么联系？没错，在 Python 虚拟机处理异常的流程中，涉及了一个 traceback 对象，在这个对象中记录栈帧链表的信息，Python 虚拟机利用这个对象来将栈帧链表中每一个栈帧的当前状态可视化，这个可视化的结果就是图 10-13 中输出的信息。

回到我们的例子，当异常发生时，当前活动的栈帧是图 10-12 中函数 h 对应的栈帧。在 Python 虚拟机开始处理异常时，它首先的行为就是创建一个 traceback 对象，用于记录异常发生时活动栈帧的状态：

```
[ceval.c]
PyObject* PyEval_EvalFrameEx(PyFrameObject *f)
{
    ......
```

```
    for (;;) {
        //巨大的 switch 语句
        if (why == WHY_EXCEPTION) {
            //创建 traceback 对象
            PyTraceBack_Here(f);
            if (tstate->c_tracefunc != NULL)
                call_exc_trace(tstate->c_tracefunc, tstate->c_traceobj, f);
        }
        ......
    } //end of for(;;)
......
}
```

这里的 tstate 还是我们之前提到的那个与当前活动线程对应的线程对象，其中的 c_tracefunc 是用户自定义的追踪函数，主要用于编写 Python 的 debugger。通常情况下这个值都是 NULL，所以我们不考虑它。这里我们的重点在于考察 Python 虚拟机究竟创建了一个怎样的 traceback 对象。

```
[traceback.c]
int PyTraceBack_Here(PyFrameObject *frame)
{
    //获得线程状态对象
    PyThreadState *tstate = frame->f_tstate;
    //保存线程状态对象中现在维护的 traceback 对象
    PyTracebackObject *oldtb = (PyTracebackObject *)
    tstate->curexc_traceback;
    //创建新的 traceback 对象
    PyTracebackObject *tb = newtracebackobject(oldtb, frame);
    //将新的 traceback 对象交给线程状态对象
    tstate->curexc_traceback = (PyObject *)tb;
    Py_XDECREF(oldtb);
    return 0;
}
```

原来 traceback 对象是保存在线程状态对象之中的，我们来看看这个 tractback 对象究竟长得是个什么样：

```
[traceback.h]
typedef struct _traceback {
    PyObject_HEAD
    struct _traceback *tb_next;
    struct _frame *tb_frame;
    int tb_lasti;
    int tb_lineno;
} PyTracebackObject;
```

一看到 tb_next，我们就恍然大悟了，原来 traceback 对象也跟 PyFrameObject 对象一样，是一个链表结构。我们进一步猜测，这个 PyTracebackObject 对象的链表结构应该跟 PyFrameObject 对象的链表结构是同构的，即一个 PyFrameObject 对象应该对应一个 PyTracebackObject 对象。我们来看看这个链表是怎么产生的：

```
[traceback.c]
PyTracebackObject* newtracebackobject
(PyTracebackObject *next, PyFrameObject *frame)
{
    PyTracebackObject *tb;
    //申请内存，创建对象
    tb = PyObject_GC_New(PyTracebackObject, &PyTraceBack_Type);
    if (tb != NULL) {
        //建立链表
        tb->tb_next = next;
        tb->tb_frame = frame;
        tb->tb_lasti = frame->f_lasti;
        tb->tb_lineno = PyCode_Addr2Line(frame->f_code, frame->f_lasti);
        PyObject_GC_Track(tb);
    }
    return tb;
}
```

前面我们看到，这里的 next 正是从线程状态对象中得到的 traceback 对象，在 newtracebackobject 中，两个 traceback 对象被链接了起来。同时，在新创建的 traceback 对象中，还利用 tb_frame 与其对应的 PyFrameObject 对象建立了联系。另外，还存储了当前最后执行的一条字节码指令及其在源代码中对应的行号。还记得 PyCodeObject 对象中的那个 co_lnotab 吗，PyCode_Addr2Line 正是利用这个 co_lnotab 获得了 frame->f_lasti 所指示的字节码指令在源代码中对应的行号。

Python 虚拟机意识到有异常抛出，并创建了 traceback 对象之后，它会在当前栈帧中寻找 except 语句，以寻找开发人员指定的捕捉异常的动作，如果没有找到，那么 Python 虚拟机将退出当前的活动栈帧，并沿着栈帧链表向上回退到上一个栈帧。在图 10-12 中，即是从函数 h 对应的 PyFrameObject 对象沿着 f_back 回退到函数 g 对应的 PyFrameObject 对象。这个回退的动作在 PyEval_EvalFrameEx 的最后完成（见代码清单 10-3）。

**代码清单 10-3**

```
PyObject* PyEval_EvalFrameEx(PyFrameObject *f)
{
  ……
  for (;;) {
      //尝试捕捉异常
      if (why != WHY_NOT)//[1]
          break;
   }
   ……
   if (why != WHY_RETURN)
       retval = NULL;//[2] ：利用 retval 通知前一个栈帧有异常出现
   ……
   //[3] ：将线程状态对象中的活动栈帧设置为当前栈帧的上一个栈帧，完成栈帧回退的动作
   tstate->frame = f->f_back;
   return retval;
}
```

可以看到，如果开发人员没有提供任何捕捉异常的动作，那么程序将执行到代码清单 10-3 的[1]处，这里也是 Python 虚拟机的主 for 循环的结尾处，由于异常没有被捕捉到，why 的值仍然是 WHY_EXCEPTION，那么将会通过 break 动作跳出 Python 执行字节码的那个 for 循环。最后，由于异常没有被捕捉到，PyEval_EvalFrame 的返回值将在代码清单 10-3 的[2]处被设置为 NULL。同时，通过重新设置当前线程状态对象中活动栈帧，完成栈帧回退的动作。

PyEval_EvalFrameEx 到这里就结束了，但是一个重要的问题随之而生，PyEval_EvalFrameEx 返回到什么地方去了？嗯，答案肯定会让你迷惑，返回到 PyEval_EvalFrameEx 里去了☺。

前面我们说了，PyEval_EvalFrameEx 是 Python 虚拟机的主要实现代码。实际上，当 Python 虚拟机运行时，这个函数是会被递归调用的。从这个函数的函数名上也可以看出一些端倪，这是一个与某个 PyFrameObject 对象的执行有关的函数，既然 PyFrameObject 对象有一个链表，那么 PyEval_EvalFrameEx 也就只能通过递归与链表结构对应了。

举个例子，当 Python 虚拟机执行函数 g 时，它是在 PyEval_EvalFrameEx 中执行与 g 对应的 PyFrameObject 对象中的字节码指令序列。当在 g 中调用 h 时，Python 虚拟机为 h 创建新的 PyFrameObject 对象，同时递归调用 PyEval_EvalFrameEx，不过这回是在其中执行与 h 对应的 PyFrameObject 对象中的字节码指令序列了。所以当在 h 中发生异常，导致 PyEval_EvalFrameEx 结束时，自然要返回到与函数 g 对应的 PyEval_EvalFrameEx 中。由于在返回时，设置的 retval 为 NULL，所以 Python 虚拟机在回到与 g 对应的 PyEval_EvalFrameEx 中后再次意识到有异常产生。接下来的动作就顺利成章了。同样是创建 traceback 对象，同样是寻找程序员指定的 except，如果没有指定异常捕捉动作，那么同样也要退出与 g 对应的 PyEval_EvalFrameEx，而返回到与 f 对应的 PyEval_EvaFrameEx。

这个沿着栈帧链不断回退的过程我们称之为**栈帧展开**。在这个栈帧展开的过程中，Python 虚拟机不断创建与各个栈帧对应的 traceback 对象，并将其链接成链表。图 10-14 给出了最终所建立的 traceback 对象链表：

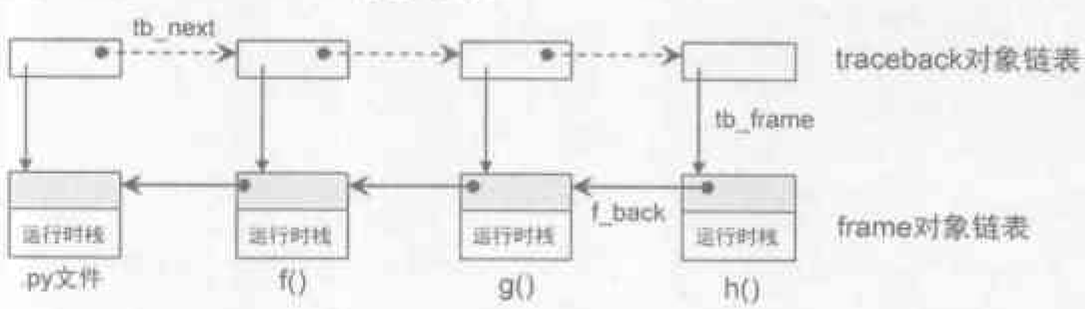



图 10-14 traceback 对象链表与 PyFrameObject 对象链表

由于我们没有设置任何的异常捕捉代码，所以最后 Python 虚拟机的执行流程会一直返回到 PyRun_SimpleFileExFlags 中。至于为什么会是这个 PyRun_SimpleFileExFlags，

我们先可以放下不管。以后在分析 Python 运行时初始化时，就可以看到这个函数的地位和作用了。

```
[pythonrun.c]
int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit,
          PyCompilerFlags *flags)
{
    ……
    //PyRun_FileExFlags 将最终调用 PyEval_EvalFrameEx
    v = PyRun_FileExFlags(fp, filename, Py_file_input, d, d, closeit, flags);
    if (v == NULL) {
        PyErr_Print();
        return -1;
    }
    ……
    return 0;
}
```

PyRun_FileExFlags 返回的值就是 PyEval_EvalFrameEx 返回的那个 NULL，所以接下来，会调用 PyErr_Print。正是在这个 PyErr_Print 中，Python 虚拟机从线程状态信息中取出其维护的 traceback 对象，并遍历 traceback 对象链表，逐个输出其中的信息，最终我们也就看到了图 10-13 所展示的异常信息。

## 10.4.2 Python 中的异常控制语义结构

### 10.4.2.1 研究对象——exception_control.py

在前面，我们细致地考察了 Python 的异常在虚拟机的级别上是什么东西，抛出异常这个动作在虚拟机的级别上对应的是什么行为；最后，我们还剖析了 Python 在处理异常时的栈帧展开行为。但遗憾的是，在前面我们只是考察了 Python 虚拟机中内建的处理异常的动作，并没有使用 Python 语言中提供的异常控制结构。在本节中，我们将通过对 exception_control.py 的剖析研究 Python 语言所提供的异常控制结构如何影响 Python 虚拟机的异常处理流程。

```
[exception_control.py]
try:
0  SETUP_FINALLY          49 (to 52)
3  SETUP_EXCEPT           16 (to 22)
raise Exception('i am an exception')
6   LOAD_NAME              0 (Exception)
9   LOAD_CONST             0 ('i am an exception')
12  CALL_FUNCTION          1
15  RAISE_VARARGS          1
18  POP_BLOCK
19  JUMP_FORWARD          26 (to 48)
except Exception, e:
22  DUP_TOP
```

```
23 LOAD_NAME            0 (Exception)
26 COMPARE_OP           10 (exception match)
29 JUMP_IF_FALSE        14 (to 46)
32 POP_TOP
33 POP_TOP
34 STORE_NAME           1 (e)
37 POP_TOP
    print e
38 LOAD_NAME            1 (e)
41 PRINT_ITEM
42 PRINT_NEWLINE
43 JUMP_FORWARD          2 (to 48)
46 POP_TOP
47 END_FINALLY
48 POP_BLOCK
49 LOAD_CONST           1 (None)
finally:
    print 'the finally code'
52 LOAD_CONST           2 ('the finally code')
55 PRINT_ITEM
56 PRINT_NEWLINE
57 END_FINALLY
58 LOAD_CONST           1 (None)
61 RETURN_VALUE
```

开始的两条字节码指令似曾相识，其实在前面剖析 Python 的循环结构的时候，我们就曾说过，SETUP_FINALLY 和 SETUP_EXCEPT 一样，它们都是调用了 PyFrame_BlockSetup。为了方便，这里再次给出 PyFrame_BlockSetup 的相关代码：

```
[frameobject.c]
void PyFrame_BlockSetup(PyFrameObject *f, int type, int handler, int level)
{
    PyTryBlock *b;
    b = &f->f_blockstack[f->f_iblock++];
    b->b_type = type;
    b->b_level = level;
    b->b_handler = handler;
}
```

```
[frameobject.h]
typedef struct _frame {
    ......
    int f_iblock;       /* index in f_blockstack */
    PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks */
    ......
} PyFrameObject;
```

```
[frameobject.h]
typedef struct {
    int b_type;         /* what kind of block this is */
    int b_handler;      /* where to jump to find handler */
    int b_level;        /* value stack level to pop to */
} PyTryBlock;
```

事到如今，想必聪明的你一定看出来了，SETUP_FINALLY 和 SETUP_EXCEPT 两条指令

不过是从 f_blockstack 中分走两块出去，图 10-15 展示了当前 f_blockstack 的情景：

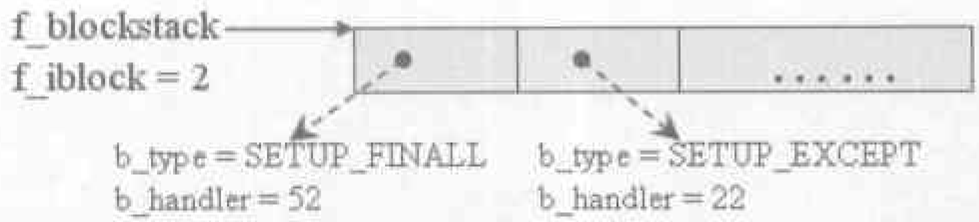



图 10-15 SETUP_FINALLY 和 SETUP_EXCEPT 完成后的 f_blockstack

在这里分出两块 PyTryBlock，肯定是要在捕捉异常时使用。不过别着急，让我们先回到抛出异常的地方："15 RAISE_VARARGS 1"。在 RAISE_VARARGS 指令之前，通过 LOAD_NAME 0、LOAD_CONST 0、CALL_FUNCTION 1 构造出了一个异常对象（CALL_FUNCTION 的剖析和对象的构造、创建并非本章关注重点，所以这里不详述，以后自有独立章节剖析，此处只需知道一个异常对象已被创建），并将此异常对象压入运行时栈中。RAISE_VARARGS 指令的工作就从把这个异常对象从运行时栈取出开始。

```
[RAISE_VARARGS]
          u = v = w = NULL;
          switch (oparg) {
          case 3:
              u = POP(); /* traceback */
              /* Fallthrough */
          case 2:
              v = POP(); /* value */
              /* Fallthrough */
          case 1:
              w = POP(); /* exc */
          case 0: /* Fallthrough */
              why = do_raise(w, v, u);
              break;
          default:
              PyErr_SetString(PyExc_SystemError,
                      "bad RAISE_VARARGS oparg");
              why = WHY_EXCEPTION;
              break;
          }
          break;
```

这里 RAISE_VARARGS 后的指令参数是 1，所以直接将异常对象取出赋给 w，然后就调用 do_raise 函数。在 do_raise 中，最终将调用之前剖析过的 PyErr_Restore 函数，将异常对象存储到当前线程的状态对象中。在 do_raise 的最后，返回了一个 WHY_EXCEPTION，这就是 why 变量的最终状态。在此之后，Python 虚拟机通过一个 break 跳出了分发字节码指令的那个巨大的 switch 语句。一旦结束了字节码指令的分发，异常的捕捉动作就有条不紊地展开了。在经过了一系列繁复的动作之后（其中包括创建并设置 traceback 对象），Python 虚拟机将携带着（why=WHY_EXCEPTION，f_iblock=2）的信息抵达真正捕捉异常的代码（见代码清单 10-4）。

代码清单 10-4

```
PyObject* PyEval_EvalFrameEx(PyFrameObject *f)
{
  ......
  for (;;) {
  ......
      while (why != WHY_NOT && f->f_iblock > 0) {
          //[1]：获得 SETUP_EXCEPT 指令创建的 PyTryBlock
          PyTryBlock *b = PyFrame_BlockPop(f);
          ......
          if (b->b_type == SETUP_FINALLY ||
              (b->b_type == SETUP_EXCEPT && why == WHY_EXCEPTION)) {
              if (why == WHY_EXCEPTION) {
                  PyObject *exc, *val, *tb;
                  //[2] ：获得线程状态对象中的异常信息
                  PyErr_Fetch(&exc, &val, &tb);
                  PUSH(tb);
                  PUSH(val);
                  PUSH(exc);
              }
              else {
                  ......
              }
              why = WHY_NOT;
              JUMPTO(b->b_handler);
              break;
          }
      }

      if (why != WHY_NOT)//[3] ：不存在异常处理代码，展开堆栈
          break;
  }
......
}
```

在代码清单 10-4 的[1]处，Python 虚拟机首先从当前的 PyFrameObject 对象中的 f_blockstack 中弹出一个 PyTryBlock 来，从图 10-15 中可以看到，弹出的这个是(b_type=SETUP_EXCEPT、b_handler=22）的 PyTryBlock。另一方面，在代码清单 10-4 的[2]处，Python 虚拟机通过 PyErr_Fetch 得到了当前线程状态对象中存储的最新的异常对象和 traceback 对象：

```
[errors.c]
void PyErr_Fetch(PyObject **p_type, PyObject **p_value, PyObject
  **p_traceback)
{
   PyThreadState *tstate = PyThreadState_GET();

   *p_type = tstate->curexc_type;
   *p_value = tstate->curexc_value;
   *p_traceback = tstate->curexc_traceback;

   tstate->curexc_type = NULL;
```

```
    tstate->curexc_value = NULL;
    tstate->curexc_traceback = NULL;
}
```

随后，Python 虚拟机将 tb、val、exc 分别压入到运行时栈中，并将 why 设置为 WHY_NOT。怎么会是 WHY_NOT 呢？不是有异常发生了吗？没错，确实应该是 WHY_NOT，在 Python 虚拟机的执行路径沿着代码清单 10-4 的[1]、[2]一路向下之后，Python 虚拟机发现了一个类型为 SETUP_EXCEPT 的 PyTryBlock 对象，已经意识到程序员在代码中已经为捕捉异常做好了准备，那么 Python 虚拟机认为自己的状态可以从 WHY_EXCEPTION 所代表的“发现异常状态”转为 WHY_NOT 代表的“正常状态”了。而接下来的处理异常的工作，则需要交给程序员指定的代码来解决，这个动作通过 JUMPTO(b->b_handler)来完成。JUMPTO 其实仅仅是进行了一下指令的跳跃，将 Python 虚拟机将要执行的下一条指令设置为异常处理代码编译后所得到的第一条字节码指令。

在 exception_control.py 中，SETUP_EXCEPT 所创建的 PyTryBlock 中的 b_handler 为 22，这时 Python 虚拟机将要执行的下一条指令就是偏移量为 22 的那条指令，从 exception_control.py 的编译结果来看，正好就是“22 DUP_POP”，异常处理代码对应的第一条字节码指令。

在处理异常的字节码指令中，有一条“26 COMPARE_OP 10”指令，这条指令将比较运行栈中存在的那个被捕捉到的异常是否跟 except 表达式中指定的异常匹配。随后通过一条进行指令跳跃的字节码指令“29 JUMP_IF_FALSE 14(to 46)”来判断是否需要进行指令跳跃。如果 COMPARE_OP 的操作结果发现异常匹配，那么 JUMP_IF_FALSE 就不会进行指令跳跃，而是接着执行“print e”表达式对应的字节码指令；而如果 COMPARE_OP 发现异常不匹配，那么 JUMP_IF_FLASE 将跳跃到偏移量为 46 的字节码指令：POP_TOP。图 10-16 清晰地展示了处理异常时所发生的不同的指令跳跃：

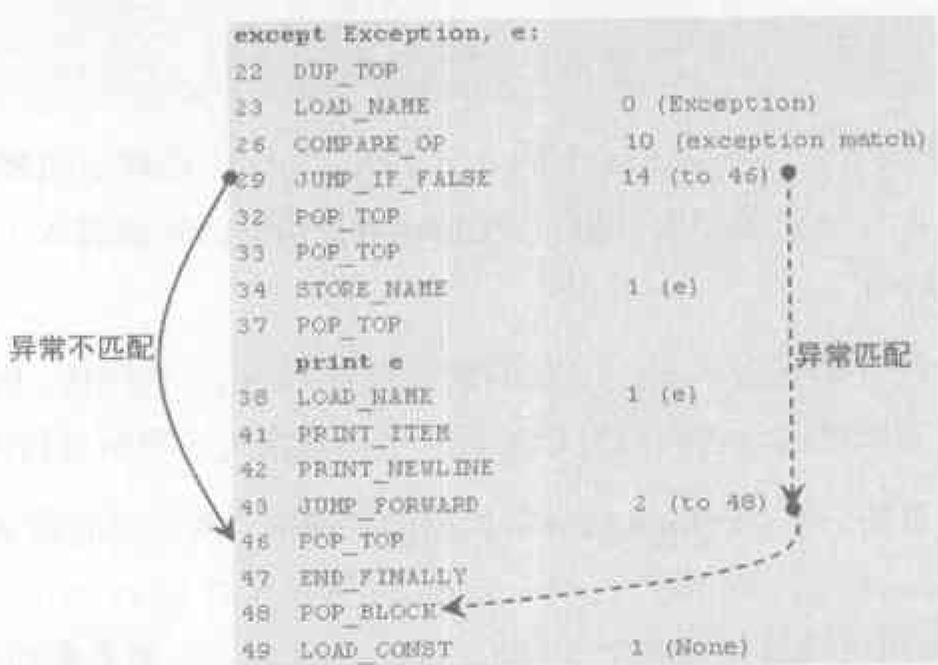



图 10-16 处理异常时的指令跳跃

我们注意到，如果忽略“print e”所对应的字节码指令，当异常不匹配时，会比异常匹配时多执行两条字节码指令代码“46 POP TOP，47 END_FINALLY”。这两条指令究竟完成怎样的工作呢，我们不妨分析一下。前面已经提到，在进入 except 表达式之前，Python 虚拟机从当前线程的状态对象中将当前的异常信息取了出来，分别为 tb、val、exc，并将其分别压入运行时栈中。当异常匹配时，它们都会从栈中取出来处理掉；如果异常不匹配，就是说虽然有 except 表达式，但是实际上并没有处理当前异常的代码。那么很显然，这时已经取出来的异常信息不能扔掉，而要重新放回到线程状态对象中，然后重新设置 why 的状态，让 Python 虚拟机的内部状态重新进入“异常发生状态”，并开始栈帧展开的动作，寻找真正能处理异常的代码。所以这两条指令就是完成这个“重返异常状态”的使命。从 END_FINALLY 指令的实现代码可以清晰地看到这一点：

```
[END_FINALLY]
            v = POP();
            if (PyExceptionClass_Check(v) || PyString_Check(v)) {
                w = POP();
                u = POP();
                PyErr_Restore(v, w, u);
                why = WHY_RERAISE;
                break;
            }
```

确实，Python 虚拟机通过 PyErr_Restore 重新设置了异常信息，并把 why 的状态设置为 WHY_RERAISE 了。

不管异常是否匹配，最终处理异常的两条岔路都会在“48 POP_BLOCK”会合：

```
[POP_BLOCK]
            {
                PyTryBlock *b = PyFrame_BlockPop(f);
                while (STACK_LEVEL() > b->b_level) {
                    v = POP();
                    Py_DECREF(v);
                }
            }
```

这里将当前 PyFrameObject 的 f_blockstack 中还剩下的那个与 SETUP_FINALLY 对应的 PyTryBlock 对象弹出，然后 Python 虚拟机的流程就进入了与 finally 表达式对应的字节码指令了。

如果一段代码只有 finally 代码块而没有 except 代码块，Python 虚拟机的行为又是怎样的呢，有兴趣的读者可以自己分析一下，其实与这里所分析的结果并无二致。

好了，总结一下，Python 的异常机制的实现中，最重要的就是 why 所表示的虚拟机状态及 PyFrameObject 对象中 f_blockstack 里存放的 PyTryBlock 对象了。变量 why 将指示 Python 虚拟机当前是否发生了异常，而 PyTryBlock 对象则指示 Python 虚拟机程序员是否为异常设置了 except 代码块和 finally 代码块。Python 虚拟机处理异常的过程就是

在 why 和 PyTryBlock 的共同作用下完成的。在图 10-17 中给出了 Python 中实现异常机制的详细的流程：

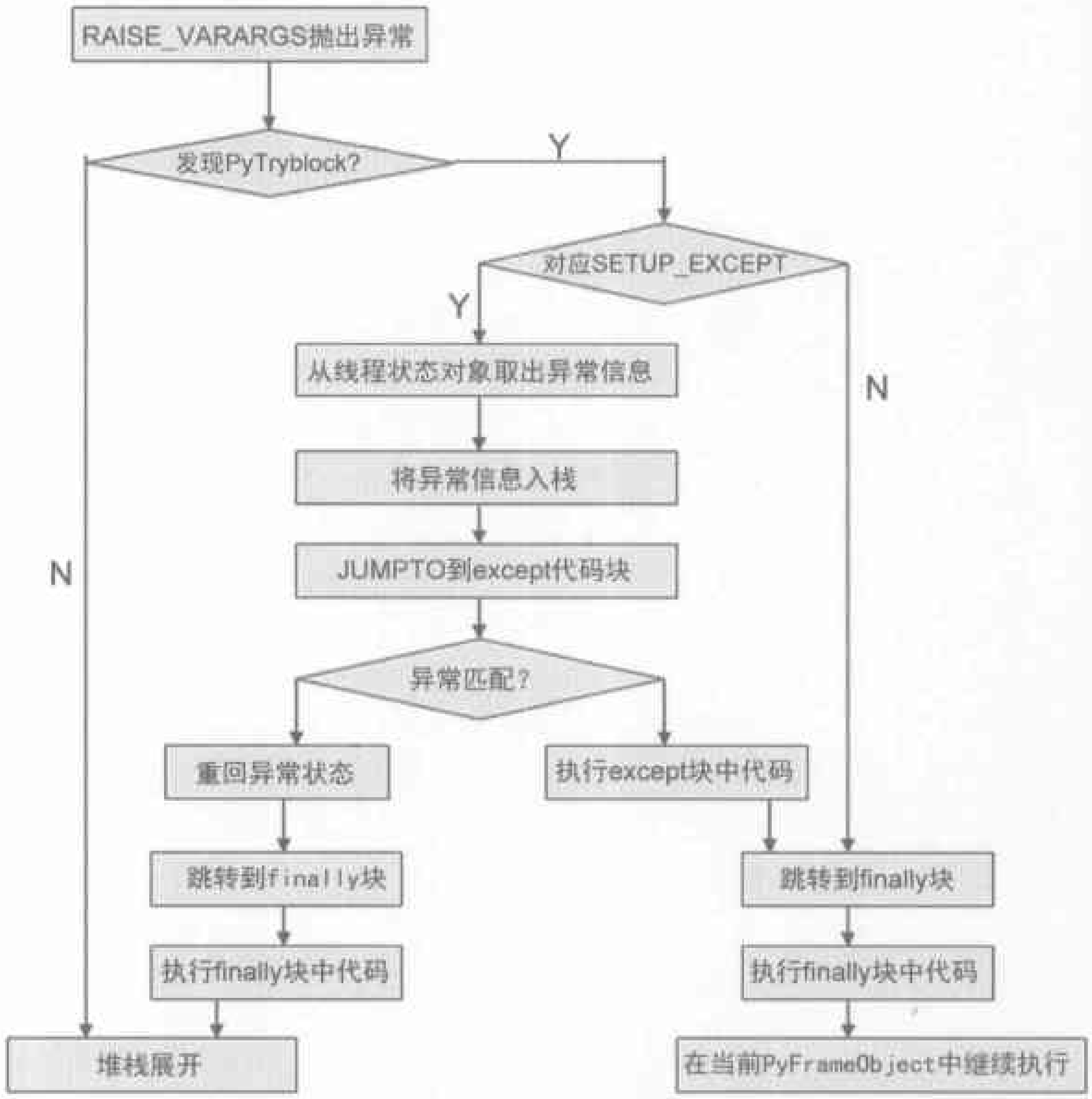



图 10-17 Python 中异常机制的流程图

CHAPTER

第 11 章

# Python 虚拟机中的函数机制

函数，作为对动作过程的抽象，是任何一门现代编程语言都具有的基本的编程元素。正是因为有了函数机制，功能分解、代码复用这些目标才能在程序中实现。之前，我们曾看过在 x86 平台上函数调用时的情形，在那里我们看到，当函数调用发生时，系统会在运行时栈中创建新的栈帧，用于函数的执行。在 Python 中这个过程也同样存在。

我们已经知道，在 Python 中，`PyFrameObject` 对象就是对一个栈帧的模拟，所以在本章我们也将看到，Python 的虚拟机在执行函数调用时会动态地创建新的 `PyFrameObject` 对象。随着函数调用链的增长，这些 `PyFrameObject` 对象之间也会连接成一条 `PyFrameObject` 对象链，这条对象链就是对 x86 平台上运行时栈的模拟。

本章将研究在 Python 中对函数的实现，以及在函数调用的过程中，Python 虚拟机的一系列动作。

## 11.1 PyFunctionObject 对象

我们说过，在 Python 中，任何东西都是一个对象，函数也不例外。在 Python 中，函数这种抽象机制是通过一个 `Python` 对象——`PyFunctionObject`——来实现的。

```
[funcobject.h]
typedef struct {
    PyObject_HEAD
    PyObject *func_code;        //对应函数编译后的 PyCodeObject 对象
    PyObject *func_globals;     //函数运行时的 global 名字空间
    PyObject *func_defaults;    //默认参数(tuple 或 NULL)
    PyObject *func_closure;     //NULL or a tuple of cell objects，用于实现 closure
    PyObject *func_doc;         //函数的文档(PyStringObject)
    PyObject *func_name;        //函数名称，函数的__name__属性，(PyStringObject)
    PyObject *func_dict;        //函数的__dict__属性(PyDictObject 或 NULL
    PyObject *func_weakreflist;
```

```
    PyObject *func_module;      //函数的__module__，可以是任何对象
} PyFunctionObject;
```

在 PyFunctionObject 对象中，并非每一个域都对我们理解函数机制有重要的意义。在本章以后的叙述中，我们将逐渐介绍重要的域，而对有些不重要的域，则会略过不谈。

在 Python 中，有两个对象都和函数有关，PyCodeObject 和 PyFunctionObject，这两个对象的区别非常重要。PyCodeObject 对象是对一段 Python 源代码的静态表示。Python 对源代码经过编译后，对一个 Code Block 会产生一个且只有一个 PyCodeObject，这个 PyCodeObject 对象中包含了这个 Code Block 的一些静态的信息，所谓静态的信息是指可以从源代码中看到的信息。比如 Code Block 中有 a = 1 这样的表达式，那么符号 a 和值 1，以及它们之间的联系就是一种静态的信息，这些信息会分别存储在 PyCodeObject 的常量表 co_consts，符号表 co_names 以及字节码序列 co_code 中，这些信息是编译时就可以得到的，因此 PyCodeObject 是编译时的结果。

而 PyFunctionObject 则不同，PyFunctionObject 对象是 Python 代码在运行时动态产生的，更准确地说，是在执行一个 def 语句的时候创建的。在 PyFunctionObject 中，当然会包括这个函数的静态信息，这些信息存储在 func_code 中，实际上，func_code 一定会指向与函数代码对应的 PyCodeObject 对象。除此之外，PyFunctionObject 对象中还包含了一些函数在执行时必需的动态信息，即上下文信息，比如 func_globals，就是函数在执行时关联的 global 作用域。global 作用域中的符号和值的对应关系必须在运行时才能确定，所以这部分必须在运行时动态创建，无法存储在 PyCodeObject 中。

对于一段 Python 代码，其对应的 PyCodeObject 对象只有一个，而代码所对应的 PyFunctionObject 对象却可能有很多个，比如一个函数多次调用，则 Python 会在运行时创建多个 PyFunctionObject 对象，每一个 PyFunctionObject 对象的 func_code 域都会关联到这个 PyCodeObject 对象。图 11-1 展示了这种 PyFunctionObject 与 PyCodeObject 对象之间的关系。

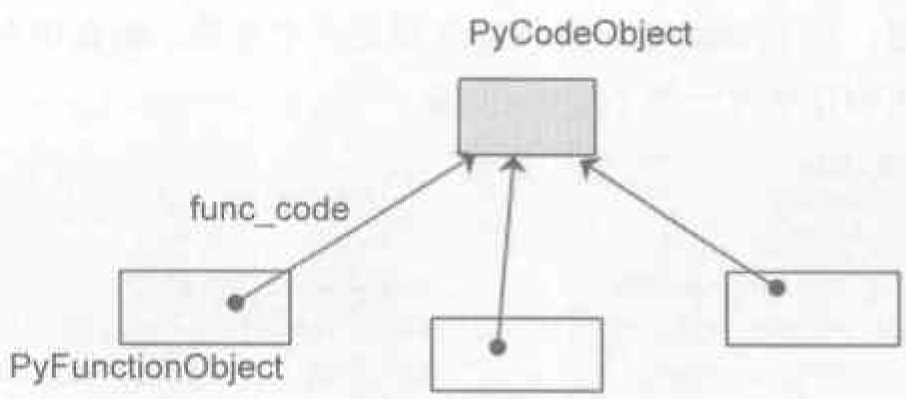



图 11-1 PyCodeObject 对象与 PyFunctionObject 对象的关系

# 11.2 无参函数调用

## 11.2.1 函数对象的创建

我们对 Python 中函数的剖析从无参函数的调用开始，因为无参函数调用是最简单的函数调用形式。从无参函数入手，我们可以在开始时不去考虑复杂的参数传递机制这样的细节，而将关注的焦点放在函数调用的整个流程框架上，以求对 Python 中的函数调用机制有一个整体框架层面的了解。我们剖析的对象是如下所示的 func_0.py，下面列出了 func_0.py 源代码及经过编译后的字节码序列。

```
[func_0.py]
def f():
0 LOAD_CONST     0 (code object f)
3 MAKE_FUNCTION 0
6 STORE_NAME     0 (f)
   print "Function"
   0   LOAD_CONST    1 ("Function")
   3   PRINT_ITEM
   4   PRINT_NEWLINE
   5   LOAD_CONST    0 (None)
   8   RETURN_VALUE

f()
9  LOAD_NAME 0 (f)
12 CALL_FUNCTION 0
15 POP_TOP
16 LOAD_CONST    1 (None)
19 RETURN_VALUE
```

以前我们考察的代码在经过 Python 编译后，都只会产生一个 PyCodeObject 对象，而在本章中，我们所考察的代码经过编译后都会产生至少两个 PyCodeObject 对象。源文件 func_0.py 经过编译后，产生的两个 PyCodeObject 对象中，一个对应整个 func_0.py，而另一个对应函数 f。这两个 PyCodeObject 对象中的常量表 co_const 和符号表 co_names 如图 11-2 和图 11-3 所示：

```
- <consts>
  + <codeObject>
    <NoneObject />
  </consts>
- <names>
    <strRef index="1" value="f" />
  </names>
  <varNames />
```

图 11-2 func_0.py 对应的 PyCodeObject 对象

```
- <consts>
    <NoneObject />
    <internStr index="0" length="8" value="Function" />
  </consts>
  <names />
  <varNames />
```

图 11-3 函数 f 对应的 PyCodeObject 对象

注意在图 11-2 的 consts 中的那个 codeObject 就是与函数 f 对应的 PyCodeObject 对象，也正是图 11-3 所示的那个 PyCodeObject 对象。在图 11-4 中，我们在 IDLE 中观察到了这两个 PyCodeObject 对象之间的联系：

```
>>> co = compile(open('func_0.py').read(), 'func_0.py', 'exec')
>>> type(co)
<type 'code'>
>>> type(co.co_consts[0])
<type 'code'>
>>> co.co_name
'<module>'
>>> co.co_consts[0].co_name
'f'
```

图 11-4 两个 PyCodeObject 对象间的关系

当 Python 虚拟机执行 func_0.py 编译后的字节码指令序列时，与我们之前所考察的字节码执行机制不同。虽然在我们所列出的代码文件中，def f() 的最后一条指令“6 STORE_NAME 0”和 print “Function”的第一条指令“0 LOAD_CONST 1”是按顺序相连的，但是实际情况并非如此。

Python 虚拟机在执行完了 def f()对应的字节码指令之后，并不会执行 print “Function”对应的字节码指令，而是会执行函数调用语句 f()所对应的第一条字节码指令，即“9 LOAD_NAME 0”。这是因为实际上在 func_0.py 编译后得到的与 func_0.py 对应的 PyCodeObject 对象中，字节码指令序列中根本就没有 print “Function”对应的字节码指令，print 语句对应的字节码指令序列是在与函数 f 对应的 PyCodeObject 对象中。只是在我们显示时，只能平面地显示字节码指令序列，不能显示出它们的层次结构。这一点需要注意。

细心的读者可能已经注意到了，在上一段的描述中，隐藏着一个惊天的秘密。按照常理来看，fun_0.py 中第 1 行 Python 代码和第 2 行 Python 代码应该是一个完整的整体，正是它们分别构成了函数的声明和函数的实现。但是在我们上一段的叙述中，看上去，函数的声明与函数的实现是分离的，甚至是分离在了不同的 PyCodeObject 对象中。没错，确实是这样，第 1 行代码虽然和第 2 行代码确实在逻辑上是一个整体，但是在 Python 实现这个函数时，却在物理上将它们分离开了。其实，第 1 行代码所对应的字节码指令序列也必须在 func_0.py 对应的 PyCodeObject 对象中，因为我们说过，在 Python 中，函数也是

一个对象。在与 func_0.py 对应的 PyCodeObject 对象中，包含着函数调用语句 f()的字节码指令序列，那么，在能调用一个函数之前，Python 必须首先创建这个函数对象。而这个函数对象的创建工作正是在 def f()这条代码处完成的。从语法上讲，def f()是函数声明语句；而从虚拟机的角度看，它其实是函数对象的创建语句。图 11-5 展现了函数机制的这种分离现象。




图 11-5 函数声明与函数定义的分离

Python 虚拟机在执行 def 语句时，会动态地创建一个函数，即一个 PyFunctionObject 对象。在这个过程中，MAKE_FUNCTION 指令是一个关键：

```
[MAKE_FUNCTION]
            v = POP(); //获得与函数 f 对应的 PyCodeObject 对象
            x = PyFunction_New(v, f->f_globals);
            Py_DECREF(v);
            ……//处理函数参数的默认值
            PUSH(x);
            break;
```

在 MAKE_FUNCTION 之前，Python 虚拟机会执行 LOAD_CONST 0。对照图 11-2，我们会看到，这条指令将函数 f 对应的 PyCodeObject 对象压入到了运行时栈中。所以，在执行 MAKE_FUNCTION 时，首先就是将这个 PyCodeObject 对象弹出运行时栈，然后以该对象和当前 PyFrameObject 对象中维护的 global 名字空间 f_globals 对象为参数，通过 PyFunction_New 创建一个新的 PyFunctionObject 对象，而这个 f_globals，将成为函数 f 在运行时的 global 名字空间（见代码清单 11-1）。

代码清单 11-1

```
[function.c]
PyObject* PyFunction_New(PyObject *code, PyObject *globals)
{
    //[1]: 申请 PyFunctionObject 对象所需的内存空间
    PyFunctionObject *op = PyObject_GC_New(PyFunctionObject, &PyFunction_Type);
    static PyObject *__name__ = 0;
    if (op != NULL) {
        //[2]: 初始化 PyFunctionObject 对象中的各个域
        ......
        //设置 PyCodeObject 对象
        op->func_code = code;
        //设置 global 名字空间
        op->func_globals = globals;
        //设置函数名
        op->func_name = ((PyCodeObject *)code)->co_name;
```

```
        //函数中的常量对象表
        consts = ((PyCodeObject *)code)->co_consts;
        //函数的文档
        if (PyTuple_Size(consts) >= 1) {
            doc = PyTuple_GetItem(consts, 0);
            if (!PyString_Check(doc) && !PyUnicode_Check(doc))
                doc = Py_None;
        }
        else
            doc = Py_None;

        ......
    }
    else
        return NULL;
    return (PyObject *)op;
}
```

在创建了 PyFunctionObject 对象之后，MAKE_FUNCTION 还会进行一些处理函数参数的动作，由于我们的 f 是一个最简单的无参函数，没有任何参数，所以在这里略过不谈。在对带参函数的剖析中，我们会深入地考察参数的传递机制。在这一节，我们的目的是建立起对 Python 函数调用过程的总体框架，尽管可能非常简陋，而且有许多细节的遗失，不过这个总体框架的建立对以后的深入考察是大有裨益的。

在 MAKE_FUNCTION 结束之后，新建的 PyFunctionObject 对象通过 PUSH 操作被压入到了运行时栈中，随后的 STORE_NAME 和 LOAD_NAME 的作用这里不再详述。到了这里，大家应该对这两条指令及其动作相当熟悉了。图 11-6 展示了 def f()语句执行完成之后运行时栈和 local 名字空间的情形。

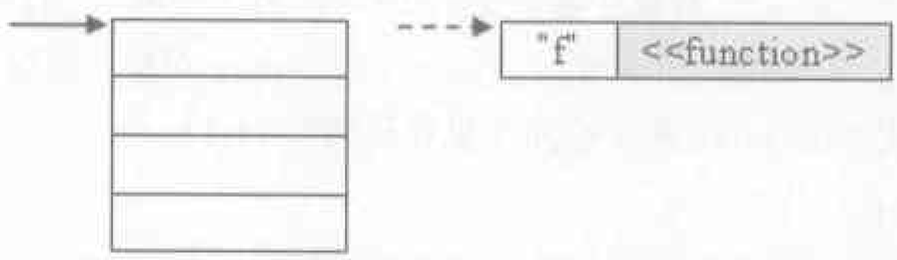



图 11-6 创建 PyFuctionObject 对象之后的虚拟机状态

## 11.2.2 函数调用

从“12 CALL_FUNCTION 0”指令开始，Python 虚拟机就进入了激动人心的函数调用动作：

```
[CALL_FUNCTION]
            PyObject **sp;
            sp = stack_pointer;
            x = call_function(&sp, oparg);
            stack_pointer = sp;
```

```
        PUSH(x);
        if (x != NULL)
            continue;
        break;
```

进入 CALL_FUNCTION 的指令代码中，Python 虚拟机虚晃一枪，在获得了当前的运行时栈栈顶指针之后，就杀入了 call_function（见代码清单 11-2）。

**代码清单 11-2**

```
[ceval.c]
static PyObject* call_function(PyObject ***pp_stack, int oparg)
{
    //[1]: 处理函数参数信息
    int na = oparg & 0xff;
    int nk = (oparg>>8) & 0xff;
    int n = na + 2 * nk;
    //[2]: 获得 PyFunctionObject 对象
    PyObject **pfunc = (*pp_stack) - n - 1;
    PyObject *func = *pfunc;
    PyObject *x, *w;

    if (PyCFunction_Check(func) && nk == 0) {
        ......
    } else {
        if (PyMethod_Check(func) && PyMethod_GET_SELF(func) != NULL) {
            ......
        }
        //[3]: 对 PyFunctionObject 对象进行调用
        if (PyFunction_Check(func))
            x = fast_function(func, pp_stack, n, na, nk);
        else
            x = do_call(func, pp_stack, na, nk);
        ......
    }
    ......
    return x;
}
```

可以看到，不光是我们的 Function，另外两个家伙，CFunction 和 Method 的调用也会进入这个 call_function，当然，这里我们把关注的焦点集中在 Function 上，不管 CFunction 和 Method。

Python 虚拟机在代码清单 11-2 的[3]处通过了 PyFunction_Check 的检查之后，就会进入 fast_function。这里需要重点关注一下 fast_function 的参数 func，同时这个 func 也是被 PyFunction_Check 进行检查的对象，我们猜想，它一定就是通过 def f() 创建的那个 PyFunctionObject 对象。

在 call_function 开始的代码清单 11-2 的[1]处，有一些处理函数参数信息的动作，现在我们不深入阐述，只需要记住，其中计算得到的 n 指明了在运行时栈中，栈顶的多少

个元素是与参数相关的。当然对于我们的 f，这里的 na、nk、n 都会是 0，所以在代码清单 11-2 的[2]处，pFunc 的值就是 pp_stack-1，因为 pp_stack 就是我们在 CALL_FUNCTION 的指令代码中传入的当前运行时栈的栈顶指针，所以很显然，*pFunc，也就是 func，指向了 Python 虚拟机在 CALL_FUNCTION 之前通过"LOAD_NAME 0"指令压入到运行时栈中的那个在 MAKE_FUNCTION 中创建的 PyFunctionObject 对象。

好了，闲话少说，我们进入 fast_function（见代码清单 11-3）。

**代码清单 11-3**

```
[ceval.c]
static PyObject *
fast_function(PyObject *func, PyObject ***pp_stack, int n, int na, int nk)
{
    PyCodeObject *co = (PyCodeObject *)PyFunction_GET_CODE(func);
    PyObject *globals = PyFunction_GET_GLOBALS(func);
    PyObject *argdefs = PyFunction_GET_DEFAULTS(func);
    PyObject **d = NULL;
    int nd = 0;

    //[1]: 一般函数的快速通道
    if (argdefs == NULL && co->co_argcount == n && nk==0 &&
        co->co_flags == (CO_OPTIMIZED | CO_NEWLOCALS | CO_NOFREE)) {
        PyFrameObject *f;
        PyObject *retval = NULL;
        PyThreadState *tstate = PyThreadState_GET();
        PyObject **fastlocals, **stack;
        int i;
        f = PyFrame_New(tstate, co, globals, NULL);
        ……
        retval = PyEval_EvalFrameEx(f, 0);
        ……
        return retval;
    }
    if (argdefs != NULL) {
        d = &PyTuple_GET_ITEM(argdefs, 0);
        nd = ((PyTupleObject *)argdefs)->ob_size;
    }
    return PyEval_EvalCodeEx(co, globals,
                (PyObject *)NULL, (*pp_stack)-n, na,
                (*pp_stack)-2*nk, nk, d, nd,
                PyFunction_GET_CLOSURE(func));
}
```

进入了 fast_function 之后，首先会抽取出 PyFunctionObject 对象中保存的 PyCodeObject 对象及函数运行时的 global 名字空间等信息。然后，我们看到，fast_function 中实际上是包含了两条执行路径，无参函数会在代码清单 11-3 的[1]处作出判断，并进入一般函数的通道，所谓一般函数，我们会在后面介绍。在一般函数的通道中，Python 虚拟机会创建新的 PyFrameObject 对象，进而调用 PyEval_EvalFrameEx，在 PyEval_EvalFrameEx 中发生的事，我们应该是很熟悉的了。

而在另一条路径上，横亘着 PyEval_EvalCodeEx。在函数 PyEval_EvalCodeEx 中，包含着一系列复杂而繁多的动作，但是如果追踪下去，在最后我们还是会看到一个 PyEval_EvalFrameEx。也就是说，无参函数进入 fast_function 之后，最终会进入 PyEval_EvalFrameEx，Python 虚拟机在一个新的 PyFrameObject（栈帧）环境中开始一次执行新的字节码指令序列的循环，这个新的字节码指令序列正是函数所对应的字节码指令序列。在我们的 func_0.py 中，也正是函数 f 中那条 print 语句对应的字节码。

从 PyEval_EvalFrameEx 开始，Python 虚拟机也就真正进入了所谓的“函数调用”的状态。这个过程实际上就是对 x86 平台上函数调用过程的模拟：创建新的栈帧，在新的栈帧中执行代码。有一点需要注意，在最终通过 PyEval_EvalFrameEx 时，PyFunctionObject 对象的影响已经消失了，真正对新栈帧产生影响的是在 PyFunctionObject 中存储的 PyCodeObject 对象和 global 名字空间。也就是说，PyFunctionObject 辛苦一场，到头来，实际上是为他人做了嫁衣裳。PyFunctionObject 主要是对字节码指令和 global 名字空间的一种打包和运输方式。

前面我们提到了一般函数这个概念，在 Python 中，什么样的函数是一般函数呢？也就是说，什么样的函数能进入快速通道呢？举个例子吧，像 C、C++、Java 中那样的函数就可以，比如 Draw(x, y)这样的函数，而像 Python 独有的函数，如 Draw(x, *key, **dict) 这样的函数则不行，Python 正是靠函数参数的形式来决定是否可以进入快速通道的。

## 11.3 函数执行时的名字空间

现在，我们对 Python 中函数调用机制的大体框架比较熟悉了，在此基础上，来看一个细节的问题。在调用 PyFunction_New 时，有一个参数是 globals，这个 globals 最终将成为与函数 f 对应的 PyFrameObject 中的 global 名字空间——f_globals。

还记得当初对 LOAD_NAME 指令的分析吗？在执行 LOAD_NAME 指令时，Python 虚拟机会依次从三个 PyDictObject 对象中进行搜索，搜索的顺序是：f_locals、f_globals、f_builtins。在 PyFunction_New 时传入的 globals 将成为在新的栈帧中执行函数 f 时的 global 名字空间。而在 MAKE_FUNCTION 的指令代码中，我们看到这个 globals 实际上就是当前 PyFrameObject 对象中的 f_globals。这就意味着，在执行 func_0.py 的字节码指令序列时的 global 名字空间和执行函数 f 的字节码指令序列时的 global 名字空间实际上是同一个名字空间。实际上这个名字空间是通过 PyFunctionObject 的携带，和字节码指令序列对应的 PyCodeObject 对象一起被传入到新的栈帧中的。图 11-7 展示了当 Python 虚拟机在执行函数 f 的字节码指令序列时，PyFrameObject 对象之间的关系及它们

与 global 名字空间的关系。

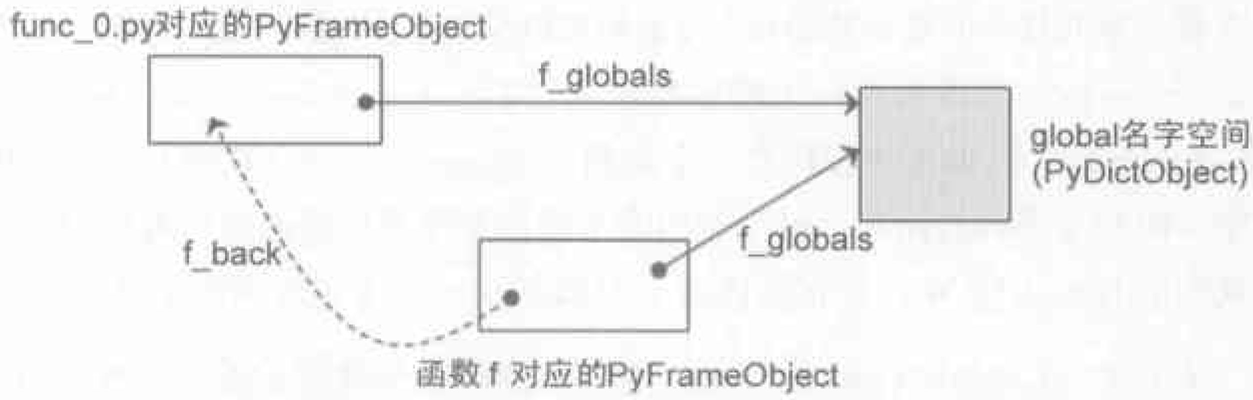



图 11-7 函数调用时 PyFrameObject 对象与 global 名字空间的关系

考虑下面一段 Python 代码，我们看看在调用 PyFunction_New 时，这个 globals 到底包含什么东西：

```
a = 1
b = 3

def f():
    print "Function f"

def g():
    print 'Function g'

f()
```

修改 Python 源代码，在 def f() 对应的 MAKE_FUNCTION 指令代码调用 PyFunction_New 之前，将 globals 的内容输出，如图 11-8 所示：

```
'a'          1
'b'          3
'__builtins__'   module
'__file__'       'func_0.py'
'__name__'       '__main__'
'__doc__'        NoneType
```

图 11-8 调用 PyFuction_New 之前的 global 名字空间

```
'a'          1
'b'          3
'g'          function
'f'          function
'__builtins__'   module
'__file__'       'func_0.py'
'__name__'       '__main__'
'__doc__'        NoneType
```

图 11-9 CALL_FUNCTION 中的 global 名字空间

可以看到，在 Python 源代码中定义的符号都包含在了这个 globals 中，这使得在函数 f 中可使用 a、b，正是依靠这个 globals 的传递，才使得函数 f 可以使用函数 f 以外的符号。

到了这里，如果你仔细观察，会发现一个致命的问题。那就是在这个 globals 中，居然没有 f 自身，这就会导致一个严重的后果，即在 f 中不能再调用函数 f，也就是说，递归函数没办法实现了。

事实真的是这样的吗？当然不是，Python 显然是支持函数的递归调用的。那么问题的关键在哪里呢？同样，在 CALL_FUNCTION 指令的开始处将已经创建的 PyFunctionObject 对象中的 func_globals 输出来看看，结果如图 11-9 所示。

可以看到，这时符号 f 已经进入了当前 func_globals，因此，符号 f 就一定会出现在函数 f 对应的那个 PyFrameObject 对象中的 f_globals 中，使得函数的递归调用得以实现。那么 f 这个狡猾的符号是在什么时候偷偷地溜进当前 PyFrameObject 对象的 f_globals 的呢？

我们回过头来看看，如果略去那个函数 g，基本上就是我们之前考察的 func_0.py，那么在 MAKE_FUNCTION 和 CALL_FUNCTION 之间只有两条字节码，STORE_NAME 和 LOAD_NAME，而 LOAD_NAME 显然没有那么大的法力，玄机正是在这个 STORE_NAME 中。前面我们分析过，STORE_NAME 是将一个符号和符号对应的值存放到当前 PyFrameObject 对象的 local 名字空间 f_locals 中。妙就妙在，当开始执行 func_0.py 时，第一次进入 PyEval_EvalFrameEx 时的那个作为参数的 PyFrameObject 对象，它的 f_locals 和 f_globals 竟然是指向同一个 PyDictObject 对象的，这一点可以在 PyFrame_New 中清晰地看到。不过仔细想一想，这样的安排其实也是理所当然的，因为在 func_0.py 外层，什么都没有了，也不会有别的作用域了。所以，当 STORE_NAME 指令将符号 f 放入 local 名字空间 f_locals 时，也就同时将 f 放进 global 名字空间 f_globals 中了。

仔细地看一看，这里实际上还隐藏着另一个有趣的结论：在 f 中我们甚至可以使用函数 g 了，虽然 g 是在 f 之后被定义的。因为在执行 f() 时，函数对象 g 已经被创建产生，并且被加入到 f_locals(f_globals) 中了，于是就可以使用了，这一点是跟 C 语言完全不同的。C 语言中函数是否可调用（可编译通过）完全是基于源代码中函数出现的位置做的分析，而 Python 则依靠的是运行时的名字空间。

## 11.4 函数参数的实现

函数，必须要配上参数才会变得有趣，并呈现出万千变化。否则，函数就会像一滩死水，会显得太死板。在这一节，我们将深入研究函数参数。上面我们已经分析了函数调用的整体框架，这一节的重点会放在参数的传递机制上。同时，在本节中，我们也将剖析函数中局部变量的实现方式。

在 Python 的函数机制中，参数一共分为四种类别的参数，我们对函数参数的剖析就从参数的类别开始。

## 11.4.1 参数类别

在 Python 中，函数的参数根据形式的不同可以分为四种类别，分别如下所示：

- 位置参数（positional argument）：`f(a, b)`、`a` 和 `b` 被称为位置参数；
- 键参数（key argument）：`f(a, b, name='Python')`，其中的 `name='Python'` 被称为键参数；
- 扩展位置参数（excess positional argument）：`def f(a, b, *list)`，其中的 `*list` 被称为扩展位置参数；
- 扩展键参数（excess key argument）：`def (a, b, **keys)`，其中的 `**key` 被称为扩展键参数。

扩展位置参数是位置参数的更高级的形式，一个扩展位置参数的存在允许我们在调用函数时向函数传入数量可变的位置参数，在某些情况下，这会给程序设计带来极大的便利。扩展键参数和键参数之间的关系也是如此。

在之前对无参函数的调用的剖析中，我们在 `call_function` 中看到了一些处理函数参数信息的操作，当时由于我们的目的在于快速建立 Python 中函数调用机制的整体框架，所以略过了。在这一节中，我们将详细地剖析 Python 在 `call_function` 中处理函数信息的操作。我们先来回顾一下 `call_function`（见代码清单 11-4）。

**代码清单 11-4**

```
[ceval.c]
static PyObject* call_function(PyObject ***pp_stack, int oparg)
{
    //[1]: 处理函数参数信息
    int na = oparg & 0xff;
    int nk = (oparg>>8) & 0xff;
    int n = na + 2 * nk;
    //[2]: 获得 PyFunctionObject 对象
    PyObject **pfunc = (*pp_stack) - n - 1;
    PyObject *func = *pfunc;
    ......
}
```

当 Python 虚拟机开始执行 `CALL_FUNCTION` 指令时，会首先获得一个指令参数 `oparg`。在这个指令参数 `oparg` 中，实际上记录的是函数参数的**个数**信息，包括位置参数的个数和键参数的个数。虽然扩展位置参数和扩展键参数是位置参数和键参数的更高级形式，但是本质上扩展位置参数是由多个位置参数构成的。这就意味着，虽然 Python 中存在四种参数形式，但是实际上我们只需要记录位置参数的个数和键参数的个数，就能知道一共有多少个参数，一共需要多大的内存空间来维护参数。

CALL_FUNCTION 指令参数的长度是两个字节，在低字节，记录着位置参数的个数，在高字节，记录着键参数的个数。因此，在理论上，Python 中的函数只能有 256 个位置参数和 256 个键参数，不过这已经足够了，估计没有哪个变态的函数会需要如此多的参数。

从 call_function 中我们可以看到 na 实际上就是位置参数的个数，而 nk 则是键参数的个数。下面我们将通过修改 Python 源代码，在 call_function 中添加输出信息，来观察拥有不同种类参数的函数中的 na 和 nk 究竟是多少。在输出 na 和 nk 的同时，我们还输出了函数对应的 PyCodeObject 对象中维护的两个与参数有关的信息：co_argcount 和 co_nlocals。

您可能觉得奇怪了，co_nlocals 看上去应该表示局部变量的个数，怎么又会和函数参数扯上关系呢？如果 co_nlocals 包含着函数参数的个数，那么 co_argcount 不就是多此一举了吗？实际上，在 Python 中，函数参数和函数的局部变量关系非常密切，在某种意义上，函数参数就是一种函数局部变量，它们在内存中是连续放置的。当 Python 需要为函数申请存放局部变量的内存空间时，就需要通过 co_nlocals 知道局部变量的总数，所以只有在 co_nlocals 中包含参数的数量，才能为参数申请内存空间。虽然 co_nlocals 中包含了参数的数量，但是没有办法从中获得这个数量，所以还必须有另外一个 co_argcount 来告诉 Python 函数一共有多少个参数。是不是觉得有点晕了？不要紧，随着我们剖析的深入，你会看到函数参数和局部变量的区别和联系。下面我们开始我们的实时观察（左边显示的是函数调用操作的指令序列）。

### 1. 位置参数

| | |
|---|---|
| LOAD_NAME 0(Py_func)<br>LOAD_CONST 1(1)<br>LOAD_CONST 2(2)<br>CALL_FUNCTION 2 | `>>> def Py_func(a, b):`<br>`        pass`<br><br>`>>> Py_func(1, 2)`<br>`[call_function] : na=2, nk=0, n=2`<br>`[call_function] : co_argcount=2, co_nlocals=2` |

### 2. 位置参数+键参数

| | |
|---|---|
| LOAD_NAME 0(Py_func)<br>LOAD_CONST 1(1)<br>LOAD_CONST 2('b')<br>LOAD_CONST 3(2)<br>CALL_FUNCTION 257 | `>>> def Py_Func(a, b):`<br>`        pass`<br><br>`>>> Py_Func(1, b=2)`<br>`[call_function] : na=1, nk=1, n=3`<br>`[call_function] : co_argcount=2, co_nlocals=2` |

从例 1 和例 2 的对比可以看出，函数参数中一个参数是位置参数还是键参数实际上仅仅是由函数实参的形式所决定的，而与函数定义时的形参没有任何关系。从例 1 到例 2，同样是为第 2 个参数 b 传递参数值 2，由于采用了不同的实参形式，就从位置参数变为了键参数。而 (na, nk) 对也从（2, 0）变为了（1, 1）。可见，na 和 nk 确实忠实地反映着位置参数和键参数的个数。

虽然在例 1 和例 2 中，na+nk 的值是一样的，都是 2，但是我们看到，n 的值却是不同的。在例 1 中，n 为 2；而在例 2 中，n 为 3。这个不同源自 n 的计算公式：n=na+2*nk。那么为什么会有这个怪异的公式呢，这一切都要从 n 的意义说起。

回到 call_function 中，之前我们就曾提到，Python 虚拟机会通过 PyObject *func = *pfunc 这条语句使 func 指向运行时栈中存放的 PyFunctionObject 对象。在这条语句之前有 pfunc = (*pp_stack)-n-1，其中 pp_stack 是当前运行时栈的栈顶指针。所以 pfunc 就是栈顶指针回退(n+1)后的结果。从例 1 和例 2 左边的指令序列可以看到，Python 虚拟机首先将 PyFunctionObject 对象压入到运行时栈，接着会将所有的与“参数有关的信息”也压入到运行时栈中，这些信息的个数会因函数的不同而不同，所以在 call_function 中，如果我们想成功地回退到运行时栈中 PyFunctionObject 的位置处，必须获得参数有关的信息的个数，这个个数正是 n。由于位置参数会导致一条 LOAD_CONST 指令，而键参数会导致两条 LOAD_CONST 指令，所以 n 的计算公式一定是 n=na+2*nk。

为什么键参数会导致两条 LOAD_CONST 指令呢？换句话说，在例 2 中传递‘b’是否必要呢？考虑一个带有默认参数值的函数 def f(a=1, b=2, c=3)，假如我们这样调用 f：f(b=4)，我们希望替换 b 的默认值，而保留 a 和 c 的默认值，如果不传递‘b’，Python 如何知道是要替换哪个变量的默认值呢？这正是键参数的作用。

3. 位置参数+扩展位置参数

| | |
|---|---|
| LOAD_NAME 0(Py_Func)<br>LOAD_CONST 1(1)<br>LOAD_CONST 2(2)<br>LOAD_CONST 3(3)<br>LOAD_CONST 4(4)<br>CALL_FUNCTION 4 | >>> def Py_Func(a, b, *list):<br>        pass<br><br>>>> Py_Func(1, 2, 3, 4)<br>[call_function] : na=4, nk=0, n=4<br>[call_function] : co_argcount=2, co_nlocals=3 |

关于扩展位置参数的使用，有一点需要特别注意。在 Python 函数的参数表之中，非键参数的位置必须在键参数之前，所以 Py_Func(1, b=2, 3, 4)这样的函数调用是非法的。从 na 的值可以看到，扩展位置参数的信息确实被归在了位置参数一类。

在第 2 行输出信息中，我们发现了一些奇特的地方，在例 1 和例 2 中，co_argcount 的值和 co_nlocals 的值是相同的，这是因为在函数内没有局部变量。但是在例 3 中，函数内同样没有局部变量，co_argcount 和 co_nlocals 的值却是不相同的。如果说 co_nlocals 的值为 3 还是合理的，那么最奇怪就是，表示函数参数个数的 co_argcount 的值居然是 2，而 Py_Func 的定义中明明声明了三个参数。唯一合理的解释是 Python 内部将扩展位置参数*list 作为一个局部变量了，这样才会有 co_argcount=2，co_nlocals=3 的结果。

我们还能看到，尽管我们调用函数时传递了四个参数，但是这丝毫不能影响 co_argcount 和 co_nlocals 的值，实际上不论这里传递的是多少个函数，都不能影响 co_argcount 和 co_locals 的值。因为 co_argcount 和 co_nlocals 是函数 Py_Func 编译后产生的 PyCodeObject 对象中的域，它们的值只可能在编译时确定。从 co_argcount=2，co_nlocals =3 的结果我们已经可以做一个大胆的猜测了，那就是，在 Python 实现 Py_Func 时，所有的扩展位置参数实际上是被存储在了一个 PyListObject 对象中。

#### 4. 位置参数+扩展键参数

| | |
|---|---|
| LOAD_NAME 0(Py_Func)<br>LOAD_CONST 1(1)<br>LOAD_CONST 2(2)<br>LOAD_CONST 3('name')<br>LOAD_CONST 4('Python')<br>LOAD_CONST 5('author')<br>LOAD_CONST 6('Guido')<br>CALL_FUNCTION 514 | >>> def Py_Func(a, b, **keys):<br>        pass<br><br>>>> Py_Func(1, 2, name='Python', author='Guido')<br>[call_function] : na=2, nk=2, n=6<br>[call_function] : co_argcount=2, co_nlocals=3 |

从 co_argcount 和 co_nlocals 的值可以看到，扩展键参数在 Python 内部同样也是作为一个局部变量来被对待的。

#### 5. 位置参数+局部变量

| | |
|---|---|
| LOAD_NAME 0(Py_Func)<br>LOAD_CONST 1(1)<br>LOAD_CONST 2(2)<br>CALL_FUNCTION 2 | >>> def Py_Func(a, b):<br>        c = 1<br><br><br>>>> Py_Func(1, 2)<br>[call_function] : na=2, nk=0, n=2<br>[call_function] : co_argcount=2, co_nlocals=3<br>>>> |

这里 co_nlocals=3 是理所当然的，因为 Py_Func 内确实含有一个局部变量了。从左边的指令序列可以看到，Python 在调用函数时，没有任何涉及到局部变量的指令。这无疑也是正确的，因为我们在图 11-5 中已经看到了，局部变量是属于另一个 PyCodeObject 的。

## 11.4.2　位置参数的传递

前面我们已经分析了无参函数的调用过程，在这一节中，我们要在此基础上更进一步，看一看带参函数的调用过程。在带参函数的调用过程中，基本的调用流程是与无参函数一样的，而重要的不同之处就在于，在调用带参函数时，Python 虚拟机必须传递参数。在这一节中，我们将重点剖析位置参数的传递过程，对于键参数的传递会在后面的小节进行剖析。

我们通过对 func_1.py 的剖析来研究位置参数的传递，同时，这个文件也用于下一小节剖析函数执行时如何对函数参数进行访问。

```
[func_1.py]
def f(name, age):
0   LOAD_CONST   0 (code object f)
3   MAKE_FUNCTION   0
6   STORE_NAME   0 (f)

    age += 5
    0   LOAD_FAST   1 (age)
    3   LOAD_CONST  1 (5)
    6   INPLACE_ADD
    7   STORE_FAST  1 (age)
    print "[", name, ", ", age, "]"
    10   LOAD_CONST  2 ('[')
    13   PRINT_ITEM
    14   LOAD_FAST   0 (name)
    17   PRINT_ITEM
    18   LOAD_CONST  3 (',')
    21   PRINT_ITEM
    22   LOAD_FAST   1 (age)
    25   PRINT_ITEM
    26   LOAD_CONST  4 (']')
    29   PRINT_ITEM
    30   PRINT_NEWLINE
    31   LOAD_CONST  0 (None)
    34   RETURN_VALUE

age = 5;
print age

f("Robert", age)
20   LOAD_NAME   0 (f)
23   LOAD_CONST  2 ('Robert')
26   LOAD_NAME   1 (age)
29   CALL_FUNCTION   2
32   POP_TOP

print age
```

编译后分别与 func_1.py 和函数 `f` 对应的 `PyCodeObject` 中的常量表和符号表如图 11-10 和图 11-11 所示：

```
- <consts>
  + <codeObject>
    <int value="5" />
    <internStr index="5" length="6" value="Robert" />
    <NoneObject />
  </consts>
- <names>
    <strRef index="4" value="f" />
    <strRef index="3" value="age" />
  </names>
  <varNames />
```

图 11-10 func_1.py 的 PyCodeObject 对象

```
- <consts>
    <NoneObject />
    <int value="5" />
    <internStr index="0" length="1" value="[" />
    <str length="2" value="," />
    <internStr index="1" length="1" value="]" />
  </consts>
  <names />
- <varNames>
    <internStr index="2" length="4" value="name" />
    <internStr index="3" length="3" value="age" />
  </varNames>
```

图 11-11 func_1.py 中函数 f 的 PyCodeObject 对象

与上一节我们剖析过的 func_0.py 不同，在编译后的 func_0.py 中，在 CALL_ FUNCTION 之前，只有一个 LOAD_NAME 0，而在编译后的 func_1.py 中，在 CALL_ FUNCTION 之前，却有三条 LOAD 指令。这三条 LOAD 指令执行完成后运行时栈的情况如图 11-12 所示：

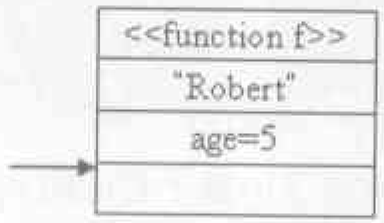



图 11-12 CALL_FUNCTION 指令执行前的运行时栈

可以看到，函数需要的参数已经被压入到了运行时栈中了。接下来执行 CALL_FUNCTION 指令，其指令参数为 2。

```
[ceval.c]
static PyObject* call_function(PyObject ***pp_stack, int oparg)
{
    int na = oparg & 0xff;
    int nk = (oparg>>8) & 0xff;
    int n = na + 2 * nk;
    PyObject **pfunc = (*pp_stack) - n - 1;
    PyObject *func = *pfunc;
    ......
}
```

前面我们提到，CALL_FUNCTION 的指令参数 oparg 中，低字节包含了位置参数的个数，所谓位置参数，就是如 f 中所见的一般的参数。而 oparg 中高字节包含了另一种参数的个数。因此 na=2，nk=0，所以 n=2。从栈顶指针 pp_stack 开始，回退 2 后，PyObject *func 正确地指向了运行时栈中存储的那个代表着 f 的 PyFunctionObject 对象。然后程序流程进入 fast_function（见代码清单 11-5）。

**代码清单 11-5**

```
[ceval.c]
static PyObject *
fast_function(PyObject *func, PyObject ***pp_stack, int n, int na, int nk)
{
    PyCodeObject *co = (PyCodeObject *)PyFunction_GET_CODE(func);
```

```
    PyObject *globals = PyFunction_GET_GLOBALS(func);

    if (argdefs == NULL && co->co_argcount == n && nk==0 &&
        co->co_flags == (CO_OPTIMIZED | CO_NEWLOCALS | CO_NOFREE)) {
        PyFrameObject *f;
        PyThreadState *tstate = PyThreadState_GET();
        PyObject **fastlocals, **stack;
        int i;
        //[1]: 创建与函数对应的 PyFrameObject 对象
        f = PyFrame_New(tstate, co, globals, NULL);
        //[2]: 拷贝函数参数: 从运行时栈到 PyFrameObject.f_localplus
        fastlocals = f->f_localsplus;
        stack = (*pp_stack) - n;
        for (i = 0; i < n; i++) {
            fastlocals[i] = *stack++;
        }
        retval = PyEval_EvalFrameEx(f, 0);
        ......
    }
    ......
}
```

代码清单 11-5 的[1]处创建了函数 `f` 对应的 `PyFrameObject` 对象，在这个过程中，函数 `f` 对应的 `PyFunctionObject` 对象中保存的 `PyCodeObject` 对象被传递给了新创建的 `PyFrameObject` 对象。随后，在代码清单 11-5 的[2]处，Python 虚拟机将参数逐个拷贝到新建的 `PyFrameObject` 对象的 `f_localsplus` 中。在分析 Python 虚拟机的框架时，我们就已经知道，这个 `f_localsplus` 所指向的内存块中也包含着 Python 虚拟机所使用的那个运行时栈。那么参数所占用的内存空间和运行时栈所占用的内存空间的关系是怎样的呢？答案就在 `PyFrame_New` 中（见代码清单 11-6）。

代码清单 11-6

```
[frameobject.c]
PyFrameObject *
PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject *globals,
        PyObject *locals)
{
    PyFrameObject *f;
    int extras, ncells, nfrees, i;

    ncells = PyTuple_GET_SIZE(code->co_cellvars);
    nfrees = PyTuple_GET_SIZE(code->co_freevars);
    extras = code->co_stacksize + code->co_nlocals + ncells + nfrees;
    ......
    //[1]: 为 f_localplus 申请 extras 的内存空间
    f = PyObject_GC_NewVar(PyFrameObject, &PyFrame_Type, extras);
    ......
    //[2]: 获得 f_localplus 中除去运行时栈外，剩余的内存数
    extras = f->f_nlocals + ncells + nfrees;
    for (i=0; i<extras; i++)
        f->f_localsplus[i] = NULL;
```

```
    f->f_valuestack = f->f_localsplus + extras;
    f->f_stacktop = f->f_valuestack;
    return f;
}
```

前面提到，在函数对应的 PyCodeObject 对象的 co_nlocals 域中，包含着函数的参数的个数，因为函数参数也是局部符号的一种。所以从 f_localsplus 开始，长度为代码清单 11-6 的[2]处计算出的 extras 的那段内存中，一定有供函数参数使用的内存。换一种说法，函数的参数存放在运行时栈之前的那片内存中。

从 PyFrame_New 创建 PyFrameObject 对象的过程中可以看到，在 f_localsplus 中，用于存储函数参数的空间和运行时栈的空间逻辑上是分离的，并不是共享同一片内存，虽然这两个空间在连续内存之内，但它们界限分明，井水不犯河水。

在处理完参数后，还没有进入 PyEval_EvalFrameEx，所以这时运行时栈还是空的。但是函数的参数已经乖乖地就位于 f_localsplus 中了。这时新建 PyFrameObject 对象中 f_localsplus 如图 11-13 所示：

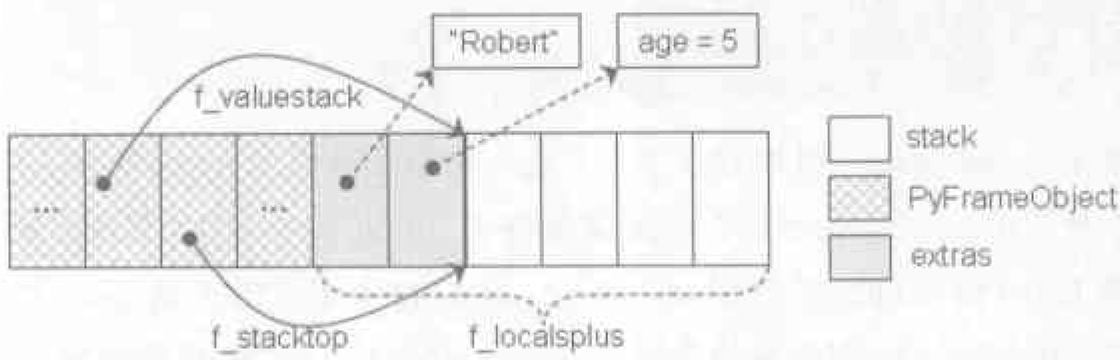



图 11-13　进入 PyEval_EvalFrameEx 之前新建 PyFrameObject 对象的内存布局

## 11.4.3 位置参数的访问

当参数拷贝的动作完成后，就会进入新的 PyEval_EvalFrameEx，开始真正的函数 f 的调用动作：

```
[func_1.py]
def f(name, age):
    age += 5
    0   LOAD_FAST    1 (age)
    3   LOAD_CONST   1 (5)
    6   INPLACE_ADD
    7   STORE_FAST   1 (age)
    print "[", name, ", ", age, "]"
......
```

劈头盖脸第一条指令就是我们未曾见过的 LOAD_FAST，而且后面还有一条 STORE_FAST，又是一对 LOAD 和 SAVE 指令，函数在被调用的过程中，对函数参数的读写动作正是通过这两条指令完成的。

```
[ceval.c]
PyObject* PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    register PyObject **fastlocals
    ……
    fastlocals = f->f_localsplus;
    ……
}

#define GETLOCAL(i) (fastlocals[i])
[LOAD_FAST]
            x = GETLOCAL(oparg);
            if (x != NULL) {
                Py_INCREF(x);
                PUSH(x);
                goto fast_next_opcode;
            }

#define SETLOCAL(i, value)  do { PyObject *tmp = GETLOCAL(i); \
                    GETLOCAL(i) = value; \
                    Py_XDECREF(tmp); } while (0)
[STORE_FAST]
            v = POP();
            SETLOCAL(oparg, v);
            goto fast_next_opcode;
```

原来，LOAD_FAST 和 STORE_FAST 这一对指令是以 f_localplus 这片内存为操作目标的。指令"0 LOAD_FAST 1"的结果是将 f_localsplus[1]中的对象压入到运行时栈中，而从图 11-13 中我们已经看到，f_localplus[1]中存放的正是 age。在完成了加法操作后，又通过 STORE_FAST 将结果放入到 f_localplus[1]中，这样就实现了对变量 age 的更新。以后在 print 中访问参数 age 时，得到的结果已经是 10 了。

现在，关于 Python 中的函数的位置参数，我们对它在函数调用过程中是如何传递，在函数执行过程中又是如何被访问，都已经真相大白了。在调用函数时，Python 将函数参数值从左至右压入到运行时栈中，在 fast_function 中，又将这些参数依次拷贝到新建的与函数对应的 PyFrameObject 对象的 f_localsplus 中。最终的效果就是，Python 虚拟机将函数调用时传入的参数，从左至右地依次存放在新建 PyFrameObject 对象的 f_localsplus 中。

在访问函数参数时，Python 虚拟机没有按照通常访问符号的做法，去查什么名字空间，而是直接通过一个索引（偏移位置）来访问 f_localsplus 中存储的符号对应的值对象。这种通过索引（偏移位置）进行访问的方法也正是"位置参数"名称的由来。

图 11-14 中详细地展示了函数在调用及执行的过程中，参数如何在 PyFrameObject 中辗转腾挪。

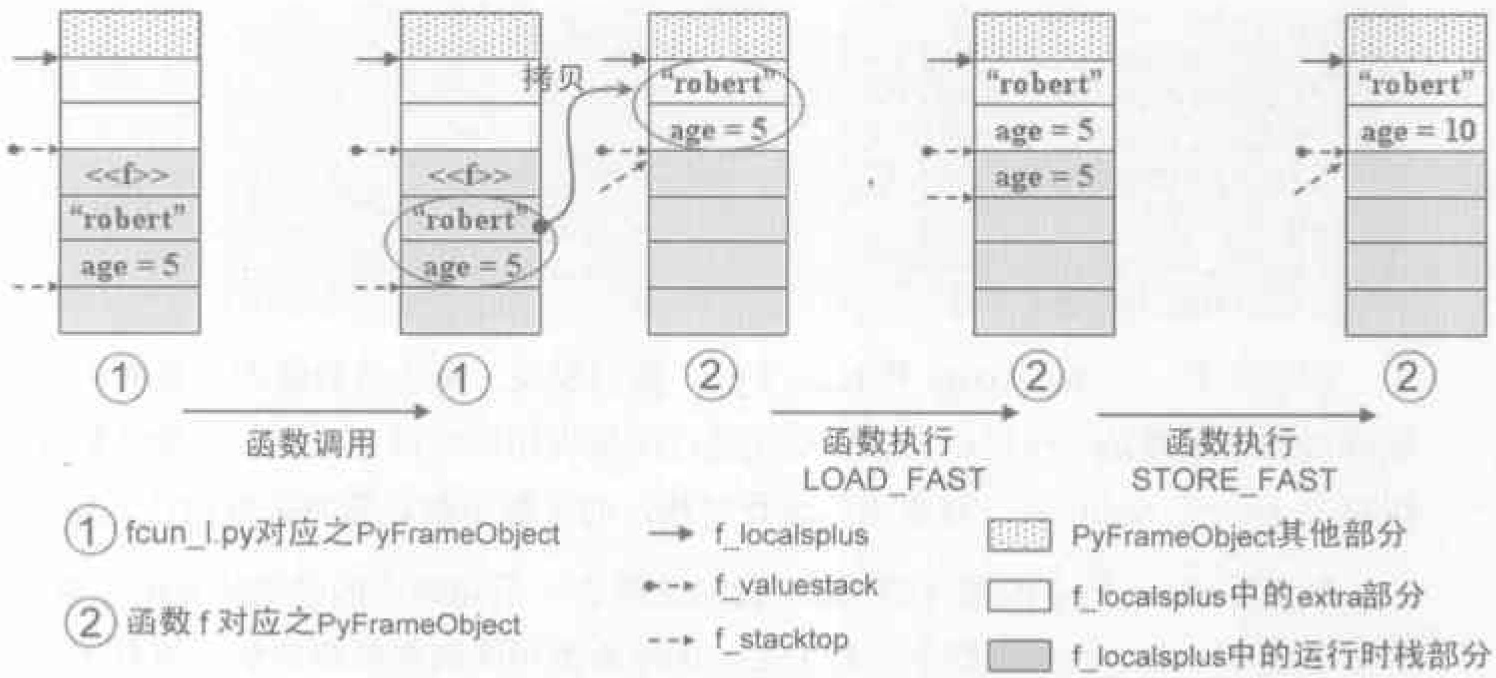



图 11-14 函数调用过程中参数的变化序列

## 11.4.4 位置参数的默认值

现在我们已经知道位置参数在函数调用的过程中是如何传递和访问的。在 Python 中，如同 C++一样，允许函数的参数有默认值。假如函数 `f` 的参数 `value` 的默认值是 1，在我们调用函数时，如果传递了 `value` 参数，那么 `f` 调用时 `value` 的值为我们传递的参数值；如果没有传递 `value` 值，那么 `f` 调用时 `value` 的值为默认值 1。那么带有默认参数值的位置参数，其实现的机制与一般的位置参数有何不同呢？这就是本节要考察的内容。我们通过 func_2.py 来深入考察 Python 中函数的默认值机制：

```
[func_2.py]
def f(a=1, b=2):
0   LOAD_CONST    0 (1)
3   LOAD_CONST    1 (2)
6   LOAD_CONST    2 (code object f)
9   MAKE_FUNCTION   2
12  STORE_NAME    0 (f)

    print a+b
    0   LOAD_FAST    0 (a)
    3   LOAD_FAST    1 (b)
    6   BINARY_ADD
    7   PRINT_ITEM
    8   PRINT_NEWLINE
    9   LOAD_CONST   0 (None)
    12  RETURN_VALUE

f()
15   LOAD_NAME    0 (f)
18   CALL_FUNCTION   0
21   POP_TOP
```

```
f(b=3)
22   LOAD_NAME      0 (f)
25   LOAD_CONST     3 ('b')
28   LOAD_CONST     4 (3)
31   CALL_FUNCTION   256
34   POP_TOP
35   LOAD_CONST     5 (None)
38   RETURN_VALUE
```

回过头看一下 func_0.py 和 func_1.py，我们发现，无论函数是否有参数，其 `def` 语句编译后的结果都是一样的，其差别是在进行函数调用的时候产生的，无参函数在调用前仅仅将 `PyFunctionObject` 对象压入运行时栈，而带参函数还需将参数也压入运行时栈。

然而在 func_2.py 的编译结果中，我们发现 `def` 语句编译的结果就显出了很大的不同，多出了两条 `LOAD_CONST` 指令，看上去，这两条语句应该与参数的默认值有关系。为了验证我们的想法，来看一看图 11-15 所示的 func_2.py 对应的 `PyCodeObject` 对象中的常量表和符号表。

```
- <consts>
    <int value="1" />
    <int value="2" />
  + <codeObject>
    <strRef index="1" value="b" />
    <int value="3" />
    <NoneObject />
  </consts>
- <names>
    <strRef index="2" value="f" />
  </names>
```

图 11-15 func_2.py 对应的常量表和符号表

再参照函数 `f` 的 `def` 语句编译后的指令序列，可以看到，开始的两条 `LOAD_CONST` 指令确实将参数的默认值压入了运行时栈，那么接下来，在 `MAKE_FUNCTION` 中，会发生什么动作呢，注意，这时 `MAKE_FUNCTION` 的参数为 2，而以前所见的 `MAKE_FUNCTION` 的参数都是 0，无论函数是否为带参函数。玄机在这里出现了（见代码清单 11-7）。

**代码清单 11-7**

```
[MAKE_FUNCTION]
        //[1]: 获得 PyCodeObject 对象，并创建 PyFunctionObject
        v = POP();
        x = PyFunction_New(v, f->f_globals);
        Py_DECREF(v);
        //[2]: 处理带默认值的函数参数
        if (x != NULL && oparg > 0) {
            v = PyTuple_New(oparg);
            while (--oparg >= 0) {
                w = POP();
                PyTuple_SET_ITEM(v, oparg, w);
            }
            err = PyFunction_SetDefaults(x, v);
```

```
            Py_DECREF(v);
        }
        PUSH(x);
```

代码清单 11-7 的[1]处创建 PyFunctionObject 对象的过程我们已经很熟悉了。在创建 PyFunctionObject 对象之后，MAKE_FUNCTION 的指令代码会处理函数参数的默认值。MAKE_FUNCTION 的指令参数表示当前在运行时栈中一共有多少个函数参数的默认值，在 func_2.py 中，这个值是 2。MAKE_FUNCTION 的指令代码会将指令参数指定的所有函数参数的默认值从运行时栈中弹出，全塞到一个 PyTupleObject 对象中。最后，通过调用 PyFunction_SetDefaults 将该 PyTupleObject 对象设置为 PyFunctionObject.func_defaults 的值。如此一来，函数参数的默认值也成了 PyFunctionObject 对象的一部分，函数和其参数的默认值最终被 Python 虚拟机捆绑在了一起，它和 PyCodeObject、global 名字空间一样，又被塞进了 PyFunctionObject 这个大包袱。

```
[funcobject.c]
int PyFunction_SetDefaults(PyObject *op, PyObject *defaults)
{
    ((PyFunctionObject *) op) -> func_defaults = defaults;
    return 0;
}
```

#### 11.4.4.1 函数 f 的第一次调用

在执行 CALL_FUNCTION，并进入 fast_function 之后，Python 的执行路径和 func_0.py，func_1.py 就不再相同了。由于在执行 MAKE_FUNCTION 指令时，Python 虚拟机已经将函数的默认参数值设置为函数对应 PyFunctionObject 对象的 func_defaults 域了，所以在代码清单 11-8 的[2]处，判断将失败，于是 Python 虚拟机将会进入 PyEval_EvalCodeEx，在进入 PyEval_EvalCodeEx 之前，将 PyFunctionObject 对象中的参数默认值信息提取了出来，并作为参数，传递给了 PyEval_EvalCodeEx。

**代码清单 11-8**

```
[ceval.c]
static PyObject *
fast_function(PyObject *func, PyObject ***pp_stack, int n, int na, int nk)
{
    PyCodeObject *co = (PyCodeObject *)PyFunction_GET_CODE(func);
    PyObject *globals = PyFunction_GET_GLOBALS(func);
    //[1]: 获得函数对应的 PyFunctionObject 中的 func_defaults
    PyObject *argdefs = PyFunction_GET_DEFAULTS(func);
    PyObject **d = NULL;
    int nd = 0;
    //[2]: 判断是否进入快速通道, argdefs != NULL 导致判断失败
    if (argdefs == NULL && co->co_argcount == n && nk==0 &&
        co->co_flags == (CO_OPTIMIZED | CO_NEWLOCALS | CO_NOFREE)) {
        ......
    }
```

```
    //这里获得函数参数默认值的信息(1.第一个默认值的地址， 2.默认值的个数)
    if (argdefs != NULL) {
        d = &PyTuple_GET_ITEM(argdefs, 0);
        nd = ((PyTupleObject *)argdefs)->ob_size;
    }
    return PyEval_EvalCodeEx(co, globals,(PyObject *)NULL,
                (*pp_stack)-n, na, //位置参数的信息
                (*pp_stack)-2*nk, nk, //键参数的信息
                d, nd, //函数默认参数的信息
                PyFunction_GET_CLOSURE(func));
}
```

PyEval_EvalCodeEx 是一个非常重要的函数，在以后分析扩展位置参数和扩展键参数时，我们还会遇到它。从 fast_function 中对 PyEval_EvalCodeEx 的调用形式可以看到，Python 虚拟机在调用 PyEval_EvalCodeEx 时，同时也将位置参数的信息和键参数的信息传递了进去。至于这些信息有什么用，我们这里先按下不表，以后会详细考察。图 11-16 展示了 func_2.py 执行时，第一次调用 f 时，Python 虚拟机所维护的 na、nk、n 等关键变量的值。

```
>>> def Py_Func(a=1, b=2):
        pass

>>> Py_Func()
[call_function] : na=0, nk=0, n=0
[call_function] : co_argcount=2, co_nlocals=2
```

图 11-16 在 func_2.py 中第一次调用 f 时的函数参数信息

回到我们对函数参数默认值的剖析中，在下面列出的 PyEval_EvalCodeEx 代码中，其函数参数中，argcount 其实就是 na 的值，而 kwcount 就是 nk 的值，当然这里都是 0。PyEval_EvalCodeEx 同样也在代码清单 11-8 的[1]处创建了新的 PyFrameObject 对象，然后，参数的默认值被直接写入到新的 PyFrameObject 对象的 f_localsplus 中（见代码清单 11-9）。

代码清单 11-9

```
[ceval.c]
PyObject *
PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
        PyObject **args, int argcount, //位置参数的信息
        PyObject **kws, int kwcount, //键参数的信息
        PyObject **defs, int defcount,  //函数默认参数的信息
        PyObject *closure)
{
    register PyFrameObject *f;
    register PyObject *retval = NULL;
    register PyObject **fastlocals, **freevars;
    PyThreadState *tstate = PyThreadState_GET();
    PyObject *x, *u;
    //[1]: 创建 PyFrameObject 对象
    f = PyFrame_New(tstate, co, globals, locals);
```

```
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + f->f_nlocals;

    if (co->co_argcount > 0 || co->co_flags & (CO_VARARGS | CO_VARKEYWORDS)) {
        int i;
        //n 为 CALL_FUNCTION 的参数指示的传入的位置参数个数，即 na，这里为 0
        int n = argcount;
        ……
        //[2]: 判断是否使用参数的默认值
        if (argcount < co->co_argcount) {
            //m = 位置参数总数 - 被设置了默认值的位置参数个数
            int m = co->co_argcount - defcount;
            //[3]: 函数调用者必须传递一般位置参数的参数值
            for (i = argcount; i < m; i++) {
                if (GETLOCAL(i) == NULL) {
                    goto fail;
                }
            }
            //[4]: n>m 意味着调用者希望替换一些默认位置参数的默认值
            if (n > m)
                i = n - m;
            else
                i = 0;
            //[5]: 设置默认位置参数的默认值
            for (; i < defcount; i++) {
                if (GETLOCAL(m+i) == NULL) {
                    PyObject *def = defs[i];
                    Py_INCREF(def);
                    SETLOCAL(m+i, def);
                }
            }
        }
    }

    retval = PyEval_EvalFrameEx(f, 0);
    return retval;
}
```

在对默认参数的讨论中，我们将位置参数继续细分为两类：一般位置参数和默认位置参数。**默认位置参数**是指指定了默认值的位置参数，而没有指定默认值的则称为**一般位置参数**。

在代码清单 11-9 的[2]处，Python 虚拟机完成了是否需要设置默认参数值的判断，当调用函数传递的位置参数的个数小于函数编译后的 `PyCodeObject` 对象中 `co_argcount` 指定的参数个数时，说明 Python 虚拟机需要为函数设定默认参数。

代码清单 11-9 的[3]处的判断是为了保证一般位置参数在函数被调用时，由调用者传递了参数值，这里的 `m` 就是我们前面所说的一般位置参数的个数。

代码清单 11-9 的[4]处确定要从哪个默认位置参数开始设定参数的默认值。考虑函数 `def g(a, b, c=1, d=2)`，如果调用函数时是如下的调用形式 `g(3, 3, 3)`，那么就不能

为参数 c 设置默认参数了，只能对 d 设置默认参数，由于 n 代表了函数调用时传递的位置参数的个数，而 m 表示一般位置参数的个数，那么 n-m 就指示了在函数调用时传递的参数中，有多少个参数不是用于一般位置参数的，那这些参数自然是用于默认位置参数的。于是这些默认位置参数不需要再设置默认值了。

当最终需要设置默认值的参数个数确定之后，Python 虚拟机会在代码清单 11-9 的[5]处从 PyFrameObject 对象的 func_defaults 中将这些参数取出，并通过 SETLOCAL 将其放入 PyFrameObject 对象的 f_localsplus 所管理的内存块中。在[5]处，i 指示了需要在 f_localsplus 中设置默认值的位置，这个 i 的值有一点值得注意，它从第一个需要设置默认值的默认位置参数的位置开始，依次向后，而这个位置之前的参数都不用设置默认值，这和 Python 中设置函数参数默认值的规则是一致的，即：函数参数的默认值从函数参数列表的最右端开始，必须连续设置。

### 11.4.4.2 函数 f 的第二次调用

在对 f 进行第二次调用时，我们重新设置了参数 b 的值，以此来观察当我们在调用函数时，为默认位置参数传递了一个参数值后，Python 虚拟机是如何用我们传递的值来替换默认值的。第二次调用函数 f 时，调用形式中的参数形式和键参数的形式是一致的。所以在 CALL_FUNCTION 之前，Python 虚拟机将 PyStringObject 对象“b”和 PyIntObject 对象 3 依次压入了运行时栈。同时，CALL_FUNCTION 的指令参数变为了 256，我们现在已经知道，这意味着 na=0，而 nk=1，这一点在图 11-17 中非常清晰地展示了出来。

```
>>> def Py_Func(a=1, b=2):
        pass

>>> Py_Func(b=3)
[call_function] : na=0, nk=1, n=2
[call_function] : co_argcount=2, co_nlocals=2
```

图 11-17 在 func_2.py 中第二次调用 f 时的函数参数信息

在 fast_function 中，Python 虚拟机同样不会选择快速通道，而是会进入 PyEval_EvalCodeEx。之前我们已经说过，PyEval_EvalCodeEx 的参数中，argcount 其实就是 na 的值，而 kwcount 其实就是 nk 的值。我们在图 11-18 中更细致地展现了当 Python 虚拟机进入 PyEval_EvalCodeEx 时各个参数的意义。

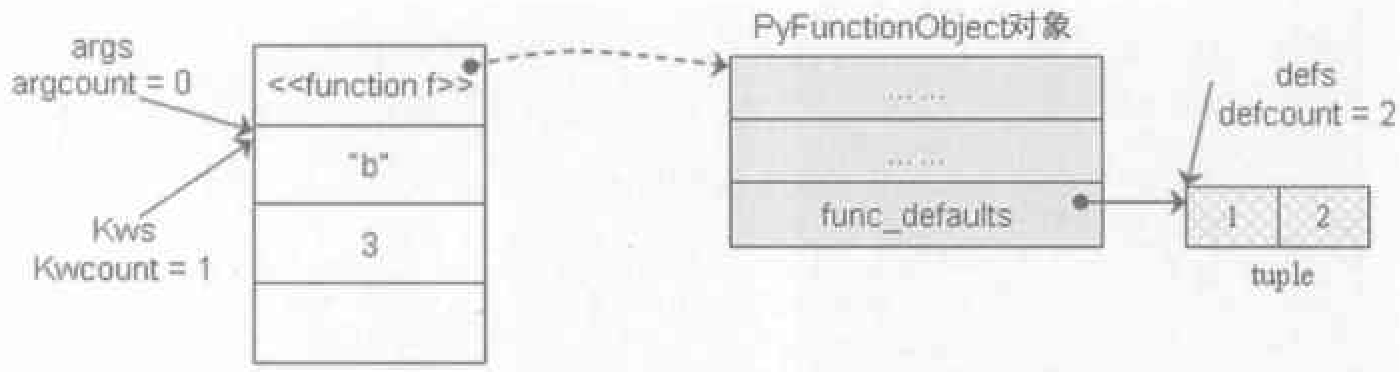



图 11-18 PyEval_EvalCodeEx 调用时各参数的意义

现在只剩最后一个关键问题了，在 PyEval_EvalCodeEx 中，3 是如何取代 b 的原始默认值 2 的呢。当然，这一切都和 kws 这家伙密切相关（见代码清单 11-10）。

代码清单 11-10

```
[ceval.c]
PyObject *
PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
        PyObject **args, int argcount,
        PyObject **kws, int kwcount,
        PyObject **defs, int defcount,
        PyObject *closure)
{
    ......
    if (co->co_argcount > 0 ||co->co_flags & (CO_VARARGS | CO_VARKEYWORDS)) {
        int i;
        int n = argcount;
        ......
        //[1]: 遍历键参数，确定函数的 def 语句中是否出现了键参数的名字
        for (i = 0; i < kwcount; i++) {
            PyObject *keyword = kws[2*i];
            PyObject *value = kws[2*i + 1];
            int j;
            //[2]: 在函数的变量名表中寻找 keyword
            for (j = 0; j < co->co_argcount; j++) {
                PyObject *nm = PyTuple_GET_ITEM(co->co_varnames, j);
                int cmp = PyObject_RichCompareBool(keyword, nm, Py_EQ);
                if (cmp > 0) //在 co_varnames 中找到 keyword
                    break;
                else if (cmp < 0)
                    goto fail;
            }

            //[3]: keyword 没有在变量名表中出现
            if (j >= co->co_argcount) {
                ......
            }
            //[4]: keyword 在变量名表中出现
            else {
                if (GETLOCAL(j) != NULL) {
                    goto fail;
                }
                Py_INCREF(value);
```

```
                SETLOCAL(j, value);
            }
        }
......
        //[5]: 设置默认位置参数的默认值
        for (; i < defcount; i++) {
            if (GETLOCAL(m+i) == NULL) {
                PyObject *def = defs[i];
                Py_INCREF(def);
                SETLOCAL(m+i, def);
            }
        }
}
```

这里算法的基本思想是：在编译时，Python 会将函数的 def 语句中出现的参数的名称都记录在变量名表(co_varnames)中。由于我们已经看到，在 f(b=3) 的指令序列中，Python 虚拟机在执行 CALL_FUNCTION 指令前会将键参数的名字压入运行时栈，那么我们在 PyEval_EvalCodeEx 中就能利用运行时栈中保存的键参数的名字在 Python 编译时得到的 co_varnames 中进行查找。最妙的是，在 co_varnames 中记录的变量名的顺序与在函数的 def 语句中出现的参数的顺序是一致的。而且我们知道，在 PyFrameObject 对象的 f_localsplus 所维护的内存中，用于存储函数参数的内存也是按照 def 语句中出现的参数的顺序排列的。所以在 co_varnames 中搜索到键参数名字后，我们可以根据所得到的序号信息直接设置 f_localsplus 中的内存，这就为默认位置参数设置了函数调用者希望的值。下面我们结合函数 f 的第二次调用来看看这个过程。

在代码清单 11-10 的[1]处的 for 循环中，i 为 0 时，就有 keyword 为 PyStringObject 对象 “b”，而 value 为 PyIntObject 对象 3。在代码清单 11-10 的[2]处的 for 循环中，会在函数 f 对应的 PyCodeObject 对象中的 co_varnames 中查找 “b”，图 11-19 显示了函数 f 编译后的 PyCodeObject 对象中的变量名表(co_varnames)。

```
- <varNames>
    <internStr index="0" length="1" value="a" />
    <internStr index="1" length="1" value="b" />
  </varNames>
```

图 11-19 函数 f 的变量名表

显然，在序号 1 处，这个查找动作将成功。而这时 j 的值正是这个序号 1。在图 11-17 中我们看到，f 对应的 PyCodeObject 对象中的 co_argcount 为 2，所以判断 j >= co->co_argcount 将不成立，最后终于来到我们苦苦寻找的地方，在代码清单 11-10 的[4]处的 else 分支中，通过 SETLOCAL，将新建的 PyFrameObject 对象中的 f_localsplus 中参数 b 对应的位置设置为 3。

代码清单 11-10 的[5]处的 for 循环是我们在 11.4.1 节中就看到过的那个为需要设置默认值的默认位置参数设置默认值的 for 循环，值得注意的是，这个 for 循环中设置函数参

数的默认值的动作只有在条件 GETLOCAL(m+i) == NULL 成立的情况下才能发生，对于 f_localsplus[1]这个位置所代表的参数 b 已经被设置为 3 了，所以不会再次设置为 b 原始的默认参数 2。

至此，Python 中函数的默认参数机制终于大白于天下。

在代码清单 11-10 的[3]处，我们看到一个奇特的判断，在什么情况下会出现 j > co->co_argcount 呢？仔细想一想就会发现，只有在 co_varnames 中搜索键参数的名字失败时才会出现这种情况。这种情况意味出现了一个键参数，这个键参数的名字没有在函数的 def 语句中出现，显然，答案已经呼之欲出了，它一定就是扩展键参数。下一节我们就将进入对扩展键参数的剖析。

## 11.4.5 扩展位置参数和扩展键参数

在 11.4.1 节的例 3 和例 4 中，我们看到了使用扩展位置参数和扩展键参数时指示参数个数的变量的值。在那里，我们发现，在函数内部没有使用局部变量时，co_nlocals 和 co_argcount 的值已经不再相同了。从它们的差异我们猜测，扩展位置参数*list 和扩展键参数**key，实际是作为一个局部变量来实现的。同时，我们还猜测，在 Python 内部，*list 是由 PyTupleObject 实现的，而**key 是由 PyDictObject 对象实现的。在本节中，将深入地剖析 Python 是如何实现扩展位置参数和扩展键参数的。

图 11-20 展示了我们考察的例子。现在我们对图 11-20 中的函数调用能编译出什么样的指令序列应该很熟悉了，这里就不再列出，仅仅考察与参数相关的重要变量的值。

```
>>> def Py_Func(value, *lst, **keys):
        pass

>>> Py_Func(-1, 1, 2, a=3, b=4)
[call_function] : na=3, nk=2, n=7
[call_function] : co_argcount=1, co_nlocals=3
```

图 11-20 带扩展位置参数和扩展键参数的例子

Python 虚拟机的执行路径最终将进入 PyEval_EvalCodeEx，这个函数我们之前已经看过了几次，对其中一些变量已经有些熟悉了。下面我们先来看看对扩展位置参数的处理（见代码清单 11-11）。

代码清单 11-11

```
[ceval.c]
PyObject * PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject
  *locals,
      PyObject **args, int argcount, //位置参数相关信息
      PyObject **kws, int kwcount, //键参数相关信息
      PyObject **defs, int defcount, //函数默认值相关信息
```

```
        PyObject *closure)
{
    register PyFrameObject *f;
    register PyObject **fastlocals, **freevars;
    PyThreadState *tstate = PyThreadState_GET();
    PyObject *x, *u;
    //创建 PyFrameObject 对象
    f = PyFrame_New(tstate, co, globals, locals);
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + f->f_nlocals;
    //[1]: 判断是否需要处理扩展位置参数或扩展键参数
    if (co->co_argcount > 0 || co->co_flags & (CO_VARARGS | CO_VARKEYWORDS)) {
        int i;
        int n = argcount;
        //这里: argcount=na=3, co_argcount=1
        if (argcount > co->co_argcount) {
            n = co->co_argcount;
        }
        //[2]: 设置位置参数的参数值
        for (i = 0; i < n; i++) {
            x = args[i];
            SETLOCAL(i, x);
        }
        //[3]: 处理扩展位置参数
        if (co->co_flags & CO_VARARGS) {
            //[4]: 将 PyTupleObject 对象放入到 f_localsplus 中
            u = PyTuple_New(argcount - n);
            SETLOCAL(co->co_argcount, u);
            //[5]: 将扩展位置参数放入到 PyTupleObject 中
            for (i = n; i < argcount; i++) {
                x = args[i];
                PyTuple_SET_ITEM(u, i-n, x);
            }
        }
    }
}
```

当 Python 在编译一个函数时，如果在其形式参数中发现了`*lst`这样的扩展位置参数的参数形式，那么 Python 会在所编译得到的 `PyCodeObject` 对象的 `co_flags` 中添加一个标识符号：`CO_VARARGS`，表示该函数在被调用时需要处理扩展位置参数。同样，对于函数的形式参数中包含`**key`这样的参数的函数，Python 将在 `co_flags` 中添加 `CO_VARKEYWORDS` 标识。所以对于含有扩展位置参数和扩展键参数的函数，代码清单 11-11 的[1]处的判断都将成立。

前面我们已经知道，在 `PyEval_EvalCodeEx` 中，`argcount` 其实就是 `na` 的值，一旦 `argcount>co->co_argcount` 成立，就意味着函数调用时传递了扩展位置参数。在代码清单 11-11 的[2]处设置了正规的位置参数后，就会进入代码清单 11-11 的[3]处对扩展位置参数的处理过程。

和我们之前的猜想相符，Python 虚拟机首先在代码清单 11-11 的[4]处创建了一个 PyTupleObject 对象，然后在代码清单 11-11 的[5]处将所有的扩展位置参数一股脑塞进这个 PyTupleObject。然后，还有最关键的一步，在[4]处，这个 PyTupleObject 对象也被 Python 虚拟机通过 SETLOCAL 放到了 PyFrameObject 对象的 f_localsplus 中，且放置的位置是 co->co_argcount。没错，正是正规的位置参数列表后的第一个位置。

了解了扩展位置参数的传递机制之后，对于扩展键参数的传递机制，实际上真的是呼之欲出了。照猫画虎，我们其实已经可以自己写出一个扩展键参数的传递机制了。不过，我们还是来看看 Python 虚拟机是如何做的（见代码清单 11-12）。

代码清单 11-12

```
[ceval.c]
PyObject * PyEval_EvalCodeEx(......)
{
    ......
    if (co->co_argcount > 0 || co->co_flags & (CO_VARARGS | CO_VARKEYWORDS)){
        int i;
        int n = argcount;
        PyObject *kwdict = NULL;
        ......
        //[1]: 创建 PyDictObject 对象，并将其放到 f_localsplus 中
        if (co->co_flags & CO_VARKEYWORDS) {
            kwdict = PyDict_New();
            i = co->co_argcount;
            //[2]: PyDictObject 对象必须在 PyTupleObject 之后
            if (co->co_flags & CO_VARARGS)
                i++;
            SETLOCAL(i, kwdict);
        }
        //遍历键参数，确定函数的 def 语句中是否出现了键参数的名字
        for (i = 0; i < kwcount; i++) {
            PyObject *keyword = kws[2*i];
            PyObject *value = kws[2*i + 1];
            int j;
            //在函数的变量名对象表中寻找 keyword
            for (j = 0; j < co->co_argcount; j++) {
                PyObject *nm = PyTuple_GET_ITEM(co->co_varnames, j);
                int cmp = PyObject_RichCompareBool(keyword, nm, Py_EQ);
                if (cmp > 0) //在 co_varnames 中找到 keyword
                    break;
                else if (cmp < 0)
                    goto fail;
            }

            //[3]: keyword 没有在变量名对象表中出现
            if (j >= co->co_argcount) {
                PyDict_SetItem(kwdict, keyword, value);
            }
            //keyword 在变量名对象表中出现
            else {
```

```
                SETLOCAL(j, value);
            }
        }
    }
}
```

其实扩展键参数的传递机制与键参数的传递机制有很大关系，上一节在分析函数参数的默认值机制时我们已经看过了键参数的传递机制，在那里我们知道 Python 虚拟机会在函数的 PyCodeObject 对象的变量名对象表(co_varnames)中查找键参数的名字，只有在查找失败时，才能确定该键参数应该属于一个扩展键参数。

和扩展位置参数的实现一样，在代码清单 11-12 的[1]处，如我们所猜想的，Python 虚拟机创建了一个 PyDictObject 对象，并且将该对象放入到了 PyFrameObject 对象的 f_localsplus 中。值得注意的是，在代码清单 11-12 的[2]处，这个判断及其后的操作保证：当函数拥有扩展位置参数时，扩展键参数的 PyDictObject 对象在 f_localsplus 中的位置一定在扩展位置参数的 PyTupleObject 对象之后的下一个位置。

然后，对调用参数传递进来的每一个键参数，Python 虚拟机都会判断它是一般的键参数，还是扩展键参数，如果是扩展键参数，就在代码清单 11-12 的[3]处将其插入到 PyDictObject 对象中。

当图 11-20 中所示的函数调用的所有参数都传递完成之后，PyFrameObject 对象中的 f_localsplus 的情形如图 11-21 所示：

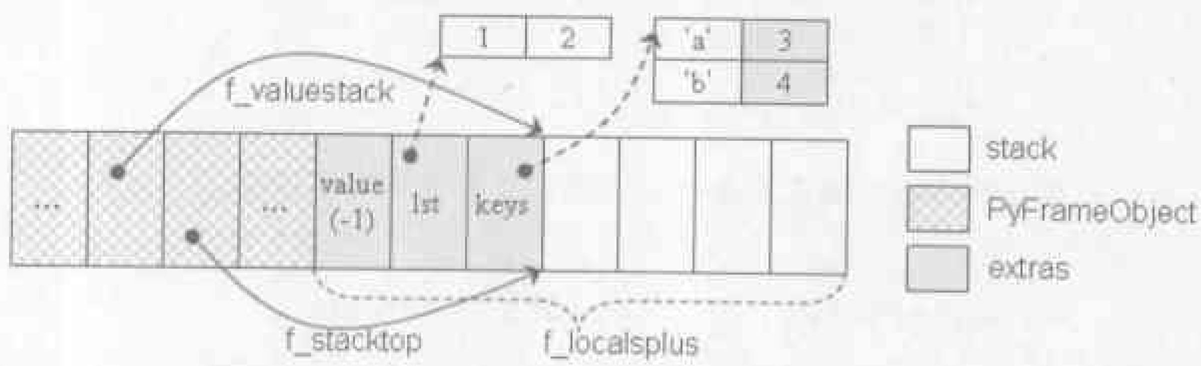



图 11-21 扩展位置参数和扩展键参数的最终归宿

当然，Python 虚拟机如何访问这些扩展位置参数和扩展键参数呢。聪明的你一定能够自己想到了☺。

## 11.5 函数中局部变量的访问

在完成了对函数参数的详细剖析之后，最后，我们来看一看，在 Python 中，函数的局部变量是如何实现的。前面提到过，函数参数实际上也是一种局部的变量，所以其实局部变量的实现机制与函数参数的实现机制是完全一致的。这个“一致”究竟意味着什么呢？

马上我们就能看到。

按照我们对 Python 的了解，当访问局部变量时，似乎应该先到 local 名字空间中去搜索变量名，但是很不幸的是，在调用函数期间，Python 虚拟机通过 PyFrame_New 创建新的 PyFrameObject 对象时，那个至关重要的 local 名字空间并没有被创建：

```
[frameobjec.c]
PyFrameObject *
PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject *globals,
        PyObject *locals)
{
  ......
    /* Most functions have CO_NEWLOCALS and CO_OPTIMIZED set. */
    if ((code->co_flags & (CO_NEWLOCALS | CO_OPTIMIZED)) ==
            (CO_NEWLOCALS | CO_OPTIMIZED))
            locals = NULL; /* PyFrame_FastToLocals() will set. */
    ......

    f->f_locals = locals;
    ......
}
```

在前面对函数调用时的 global 名字空间的剖析中，我们已经看到，当 Python 虚拟机执行***.py 时，f_locals 和 f_globals 指向的是同一个 PyDictObject，尽管有些寒碜，但毕竟还有一个 PyDictObject 对象可以使用。现在倒好，f_locals 变成了光杆司令一个，那些重要的局部符号究竟会存放在哪里呢？别急，我们先来看一个使用局部变量的函数。

```
[func_3.py]
def f(a, b):
    c = a + b
    0  LOAD_FAST   0 (a)
    3  LOAD_FAST   1 (b)
    6  BINARY_ADD
    7  STORE_FAST  2 (c)
    print c
    10 LOAD_FAST   2 (c)
......
```

一目了然，原来局部变量也是利用 LOAD_FAST 和 STORE_FAST 来操作的，更进一步，局部变量的容身之处和函数参数一样，都是在 f_localsplus 中运行时栈前面的那段内存空间中。回过头参考一下图 11-21，我们对局部变量 c 的藏身之处已经了然于胸。

为什么在函数的实现中没有使用 local 名字空间呢？这是因为函数中的局部变量总是固定不变的，所以在编译时就能确定局部变量使用的内存空间的位置，也能确定访问局部变量的字节码指令应该如何访问内存。有了这些信息，Python 就能使用静态的方法来实现局部变量，而不需要借助于动态地查找 PyDictObject 对象的技术，毕竟，函数调用实在是太普遍了，静态的方法可以极大地提高函数执行的效率。

## 11.6 嵌套函数、闭包与 decorator

在前几章我们已经提到，在 Python 中，有一个核心的概念叫名字空间，一段代码执行的结果不光取决于代码中的符号，更多地是取决于代码中符号的语义，而这个运行时的语义正是由名字空间决定的。名字空间是在运行时由 Python 虚拟机动态维护的，但是有时，我们希望能将名字空间静态化。换句话说，我们希望有的代码不受名字空间变换的影响，始终保持一致的行为和结果。这样做有什么意义呢？

来考虑一个具体的例子，假如我们想要定一个基准值，然后将许多值与这个值进行比较，最简单的方法就是写一个函数：

```
[compare1.py]
def compare(base, value):
  return value > base

compare(10, 5)
compare(10, 20)
```

我们将 10 作为基准值，然后将 5 和 20，分别与 10 进行比较，这样写当然没问题。但是如果仔细观察一下就会发现，我们不得不每次都将基准值作为参数传入函数，这一点相当不爽。Python 提供了一种方法，可以使我们在编写代码时，只设置一次基准值。这种方法利用了嵌套函数。

```
[compare2.py]
base = 1
def get_compare(base):
  def real_compare(value):
      return value > base
  return real_compare

compare_with_10 = get_compare(10)
print compare_with_10(5) //输出 False
print compare_with_10(20)  //输出 True
```

在 compare2.py 中，我们只设置了一次基准值。此后，在每次进行比较操作时,尽管调用的实际函数 `real_compare` 的 `local` 名字空间中没有 `base`，而 `global` 名字空间中有"`base = 1`"，但是函数调用的结果显示，`real_compare` 以一种神奇的方式得知了 `base` 应该为 10 而不是 1。

也就是说，在 `real_compare` 这个函数作为返回值被传递给 `compare_with_10` 的时候，有一个名字空间已经与 `real_compare` 紧紧地绑定在一起了，在执行 `real_compare` 的代码时，这个名字空间又被恢复了，这就是一种将名字空间静态化的方法。这个名字空间与函数捆绑后的结果被称为一个闭包(closure)。在前面我们已经看到了，`PyFunctionObject` 是 Python 虚拟机专门为字节码指令准备的大包袱，`global` 名字空间、默认参数都能在

PyFunctionObject 中与字节码指令捆绑在一起，同样的，PyFunctionObject 就是 Python 中闭包的具体表现了。

如果你还记得的话，关于闭包，实际上在前面介绍名字空间与作用域时，就已经遇到了。在那里，我们提到，Python 语言的设计遵循了一个核心的作用域规则——最内嵌套作用域规则，compare2.py 的结果正是这套规则产生的效果。我们同时也提到了，闭包是这个规则的一种实现方式，规则是形而上的“道”，而闭包是形而下的“器”。现在，我们马上就可以看到，不用闭包，我们同样可以实现 compare2.py 想要实现的效果。

```
[compare3.py]
base = 1
def get_compare(base):
  def real_compare(value, base=base):
      return value > base
  return real_compare

compare_with_10 = get_compare(10)
print compare_with_10(5)      //输出 False
print compare_with_10(20)     //输出 True
print compare_with_10(5, 1)  //输出 True
```

为了证明这段代码中闭包确实不存在了，我们在最后改变了比较的基准值（base），如果使用闭包，最后一行输出代码是会抛出异常的。

这个结果相当令人惊讶，利用函数的默认参数，我们实现了闭包的效果。这个现象很自然地会引起我们的猜测，闭包和默认参数的实现方式莫非是相似的？这个猜测也合情合理，毕竟，如果默认参数的实现方式可以起作用，Python 应该没有必要去实现另一套完全不同的机制，徒增复杂度。猜来猜去，闭包到底是如何实现的呢？别急，到后面自然会水落石出，现在我们先放一放，先来看看 PyCodeObject、PyFunctionObject 和 PyFrameObject 这些我们已经很熟悉的对象中，与闭包相关的属性。

## 11.6.1 实现闭包的基石

从 compare2.py 中可以发现，闭包的创建通常是利用嵌套函数来完成的。在 PyCodeObject 中，与嵌套函数相关的属性是 co_cellvars 和 co_freevars。两者的具体含义如下：

- co_cellvars：通常是一个 tuple，保存嵌套的作用域中使用的变量名集合；
- co_freevars：通常也是一个 tuple，保存使用了的外层作用域中的变量名集合。

考虑下面的代码：

```
[closure.py]
def get_func():
```

```
    value = "inner"
    def inner_func():
        print value
    return inner_func

show_value = get_func()
show_value()
```

很显然，closure.py 会编译出 3 个 PyCodeObject，其中有两个，一个与函数 get_func 对应，一个与函数 inner_func 对应，那么，与 get_func 对应的 PyCodeObject 对象中的 co_cellvars 就应该包含字符串“value”，因为其嵌套作用域（inner_func 的作用域）中使用了这个符号；同理，与函数 inner_func 对应的 PyCodeObject 对象中的 co_freevars 中应该也有字符串“value”。图 11-22 和图 11-23 分别显示了 get_func 和 inner_func 的 co_cellvars 和 co_freevars。

```
- <consts>
    <NoneObject />
    <internStr index="0" length="5" value="inner" />
  + <codeObject>
  </consts>
  <names />
- <varNames>
    <strRef index="2" value="inner_func" />
  </varNames>
  <freeVars />
- <cellVars>
    <strRef index="1" value="value" />
  </cellVars>
```

图 11-22 get_func 对应的 PyCodeObject

```
- <consts>
    <NoneObject />
  </consts>
  <names />
  <varNames />
- <freeVars>
    <internStr index="1" length="5" value="value" />
  </freeVars>
  <cellVars />
```

图 11-23 inner_func 对应的 PyCodeObject

在 PyFrameObject 对象中，也有一个属性与闭包的实现相关，这个属性就是 f_localsplus，这样一说，是不是有些隐隐约约地猜到了呢？其实在 PyFrame_New 中，就已经泄漏了天机，看下面这行代码：

```
extras = code->co_stacksize + code->co_nlocals + ncells + nfrees;
```

extras 正是 f_localsplus 指向的那片内存的大小，这里已经清清楚楚地说明了，嘿，兄弟，这片内存是属于四个家伙的：运行时栈、局部变量、cell 对象（对应 co_cellvars）和 free 对象（对应 co_freevars）。到这里 f_localsplus 的所有秘密才完全暴露了出来，

图 11-24 显示了 f_localsplus 的布局。

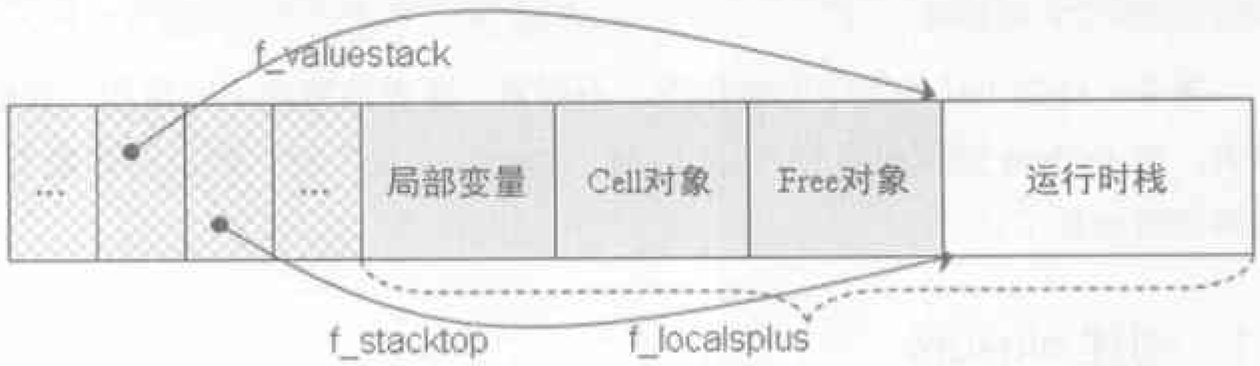



图 11-24 f_localsplus 的完整内存布局

在 PyFunctionObject 中，还有一个与闭包实现相关的属性。当然，毕竟 PyFunction-Object 是个大包嘛。这个属性，在下一节，我们就可以看到它的真身了。

## 11.6.2 闭包的实现

在介绍了实现闭包的一些基石之后，我们可以开始追踪闭包的具体实现过程了，当然，首先需要了解一下 closure.py 编译后的字节码指令序列。

```
[closure.py]
def get_func():
0    LOAD_CONST  0 (<code object get_func>)
3    MAKE_FUNCTION 0
6    STORE_NAME 0 (get_func)
   value = "inner"
   0    LOAD_CONST  1  ('inner')
   3    STORE_DEREF  0  (value)
   def inner_func():
   6    LOAD_CLOSURE  0  (value)
   9    BUILD_TUPLE  1
   12   LOAD_CONST  2  (<code object inner_func>)
   15   MAKE_CLOSURE  0
   18   STORE_FAST  0  (inner_func)
      print value
      0    LOAD_DEREF  0  (value)
      3    PRINT_ITEM
      4    PRINT_NEWLINE
      5    LOAD_CONST  0  (None)
      8    RETURN_VALUE
   return inner_func
   21   LOAD_FAST  0  (inner_func)
   24   RETURN_VALUE

show_value = get_func()
9   LOAD_NAME  0  (get_func)
12  CALL_FUNCTION 0
15  STORE_NAME  1  (show_value)
show_value()
18  LOAD_NAME  1  (show_value)
21  CALL_FUNCTION  0
```

```
24  POP_TOP
25  LOAD_CONST  1  (None)
28  RETURN_VALUE
```

乖乖，好多不认识的字节码指令。不要紧，顺着函数调用的流程，我们都能一一手到擒来。在 Python 虚拟机执行"`12 CALL_FUNCTION 0`"时，就已经开始为 closure 的实现悄悄添砖加瓦了。

### 11.6.2.1 创建 closure

前面已经看到，在 Python 虚拟机执行 `CALL_FUNCTION` 指令时，会进入 `fast_function` 函数。而在 `fast_function` 函数中，由于当前的 `PyCodeObject` 为 `get_func` 对应之 `PyCodeObject`，其中的 `co_flags` 为 `3(CO_OPTIMIZED|CO_NEWLOCALS)`，所以最终不符合进入快速通道的条件，而会进入 `PyEval_EvalCodeEx`。

图 11-22 中已经显示，当前的这个 `PyCodeObject` 的 `co_cellvars` 中是有东西的，在 `PyEval_EvalCodeEx` 中，Python 虚拟机会如同处理默认参数一样，将 `co_cellvars` 中的东西拷贝到新创建的 `PyFrameObject` 的 `f_localsplus` 中（见代码清单 11-13）。

**代码清单 11-13**

```
[ceval.c]
PyObject * PyEval_EvalCodeEx(……)
{
    ……
    if (PyTuple_GET_SIZE(co->co_cellvars)) {
        int i, j, nargs, found;
        char *cellname, *argname;
        PyObject *c;
        ……
        for (i = 0; i < PyTuple_GET_SIZE(co->co_cellvars); ++i) {
            //[1]. 获得被嵌套函数共享的符号名
            cellname = PyString_AS_STRING(PyTuple_GET_ITEM(co->co_cellvars, i));
            found = 0;
            ……//处理被嵌套函数共享外层函数的默认参数
            if (found == 0) {
                c = PyCell_New(NULL);
                if (c == NULL)
                    goto fail;
                SETLOCAL(co->co_nlocals + i, c);
            }
        }
    }
    ……
```

嵌套函数有时候会很复杂，比如内层嵌套函数引用的不是外层嵌套函数的局部变量，而是外层嵌套函数的一个拥有默认值的参数，这些复杂的情况我们就不再深入了，所以这里有些代码被删减了，但是原理都是差不多的，有兴趣的读者可参阅 Python 的源码。

在代码清单 11-13 中的[1]处，Python 虚拟机获得了被内层嵌套函数引用的符号名，在我们的例子中，就是获得了一个字符串“value”。这里的 found 需要解释一下，found 是被内层嵌套函数引用的符号是否已经与某个值绑定的标识，或者说与某个对象建立了约束关系。只有在内层嵌套函数引用的是外层函数的一个有默认值的参数时，这个标识才可能为 1，对于我们的例子，found 一定为 0。所以，Python 虚拟机接下来会创建一个 cell 对象——PyCellObject。

```
[cellobject.h]
typedef struct {
    PyObject_HEAD
    PyObject *ob_ref;   /* Content of the cell or NULL when empty */
} PyCellObject;
```

这个对象非常简单，仅仅维护了一个 ob_ref，指向一个 Pythong 中的对象，我们来看看创建 PyCellObject 对象的代码。

```
[cellobject.c]
PyObject *PyCell_New(PyObject *obj)
{
    PyCellObject *op;
    op = (PyCellObject *)PyObject_GC_New(PyCellObject, &PyCell_Type);
    op->ob_ref = obj;
    Py_XINCREF(obj);
    _PyObject_GC_TRACK(op);
    return (PyObject *)op;
}
```

在我们的例子中，创建的 PyCellObject 对象维护的 ob_ref 指向了 NULL，也就是说，现在还不知道到底是个什么东西，那什么时候才能知道呢？其实在 closure.py 中已经很明显了，就是在 value = "inner"这个赋值语句执行的时候。随后，这个 cell 对象被拷贝到了新创建的 PyFrameObject 对象的 f_localsplus 中。值得注意的是，这个对象被拷贝到的位置是 co->co_nlocals + i，说明在 f_localsplus 中，cell 对象的位置是在局部变量之后的，这完全符合图 11-24 所示的内存布局。

在处理 co_cellvars 时，有一个奇怪的地方，在我们创建 PyCellObject 对象的过程中，代码清单 11-13 的[1]处获得的 cellname 完全被忽略了。实际上，这和前面分析到的 Python 函数机制将对局部变量符号的访问方式从对 dict 的查找变为对 list 的索引是一个道理。在 get_func 函数执行的过程中，对 value 这个 cell 变量的访问将通过基于索引访问 f_localsplus 完成，因而完全不需要再知道 cellname 了。这个 cellname 实际上是在处理内层嵌套函数引用外层函数的默认参数时产生的。

在处理了 cell 对象之后，Python 虚拟机将进入 PyEval_EvalFrameEx，从而正式开始对函数 get_func 的调用过程。

首先执行的"0 LOAD_CONST 1"指令将 PyStringObject 对象"inner"压入到运行时栈，然后 Python 虚拟机开始执行一条对我们而言全新的字节码指令——STORE_DEREF。

```
[PyEval_EvalFrameEx]
freevars = f->f_localsplus + co->co_nlocals;

[STORE_DEREF]
        w = POP();
        x = freevars[oparg];
        PyCell_Set(x, w);
        Py_DECREF(w);
```

从运行时栈弹出的是 PyStringObject 对象"inner"，而从 f_localsplus 中取得的是 PyCellObject 对象，看样子，是要设置 PyCellObject 对象中的 ob_ref 啊，没错，PyCell_Set 就是干这个勾当的。

```
[cellobject.h]
#define PyCell_SET(op, v) (((PyCellObject *)(op))->ob_ref = v)

[cellobject.c]
int PyCell_Set(PyObject *op, PyObject *obj)
{
    Py_XDECREF(((PyCellObject*)op)->ob_ref);
    Py_XINCREF(obj);
    PyCell_SET(op, obj);
    return 0;
}
```

这样一来，f_localsplus 就发生了变化了，如图 11-25 所示。

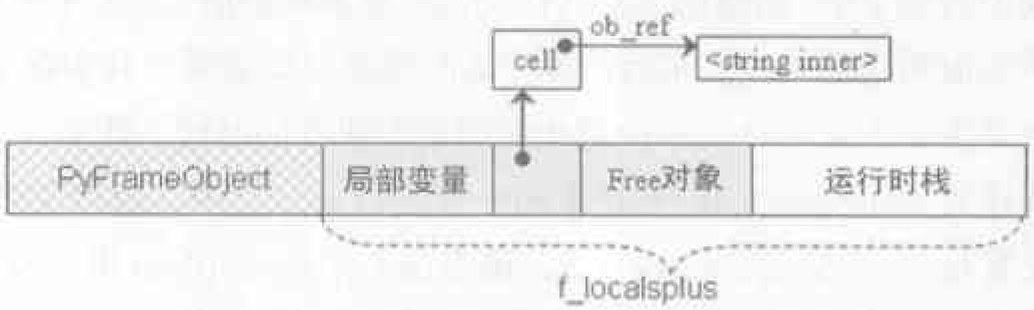



图 11-25 设置 cell 对象之后的 get_func 函数的 PyFrameObject 对象

现在在 get_func 的环境中我们知道了 value 符号对应着一个 PyStringObject 对象，但是 closure 的作用是将这个约束进行冻结，使得在嵌套函数 inner_func 被调用时还能使用这个约束。这一次，又要请 PyFunctionObject 这个邮递员出马了。在执行 closure.py 中接下来的"def inner_func()"表达式时，Python 虚拟机就会将 (value, "inner") 这个约束塞到 PyFunctionObject 中。

```
[LOAD_CLOSURE]
                x = freevars[oparg];
                Py_INCREF(x);
                PUSH(x);
```

“6 LOAD_CLOSURE 0”将刚刚放置好的 PyCellObject 对象取出，并压入运行时栈，接着的“9 BUILD_TUPLE 1”指令将 PyCellObject 对象打包进一个 tuple 中，显然，这个 tuple 中可以放置多个 PyCellObject，不过我们的例子中只有一个 PyCellObject。

随后，Python 虚拟机通过“12 LOAD_CONST 2”指令将 inner_func 对应的 PyCodeObject 对象也压入到运行时栈中，接着以一个“15 MAKE_CLOSURE 0”指令完成约束与 PyCodeObject 的绑定。

```
[MAKE_CLOSURE]
            {
                v = POP(); //获得 PyCodeObject 对象
                x = PyFunction_New(v, f->f_globals); //绑定 global 名字空间
                v = POP();//获得 tuple，其中包含 PyCellObject 对象的集合
                err = PyFunction_SetClosure(x, v); //绑定约束集合
                ……//处理拥有默认值的参数
                PUSH(x);
            }
```

表达式“def inner_func()”所对应的最后一条“18 STORE_FAST 0”指令将新创建的 PyFunctionObject 对象放置到了 f_localsplus 中，这样 f_localsplus 又发生了变化，如图 11-26 所示。

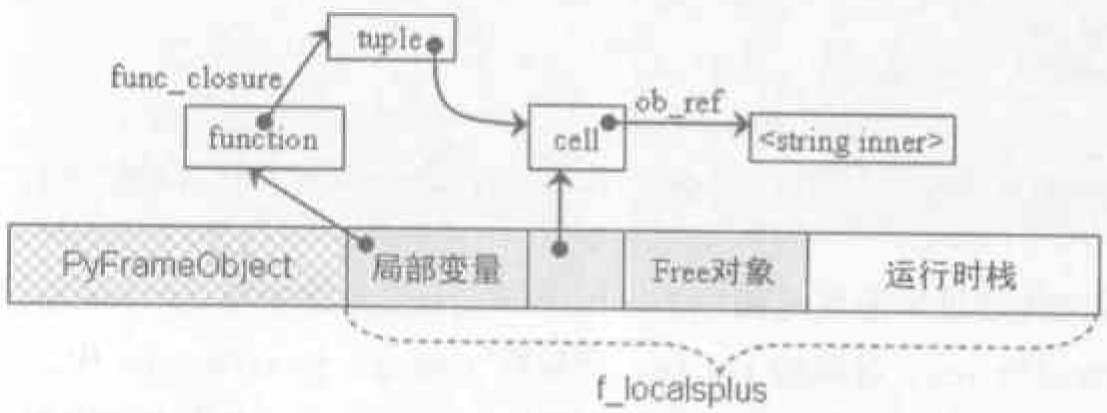



图 11-26 设置 function 对象之后的 get_func 函数的 PyFrameObject 对象

在 get_func 的最后，这个新建的 PyFunctionObject 对象作为返回值返回给了上一个栈帧，并被压入到该栈帧的运行时栈中。

#### 11.6.2.2 使用 closure

closure 是在 get_func 中被创建的，而对 closure 的使用，则是在 inner_func 中。在执行“show_value()”对应的 CALL_FUNCTION 指令时，和 inner_func 对应的 PyCodeObject 中的 co_flags 里包含了 CO_NESTED，所以在 fast_function 中不能通过快速通道的验证，从而只能进入 PyEval_EvalCodeEx。

在图 11-23 中，我们已经看到，inner_func 对应的 PyCodeObject 中 co_freevars 里有引用的外层作用域中的符号名，在 PyEval_EvalCodeEx 中，就会对这个 co_freevars 进行处理。

```
[ceval.c]
PyObject * PyEval_EvalCodeEx(......)
{
    ......
    if (PyTuple_GET_SIZE(co->co_freevars)) {
        int i;
        for (i = 0; i < PyTuple_GET_SIZE(co->co_freevars); ++i) {
            PyObject *o = PyTuple_GET_ITEM(closure, i);
            freevars[PyTuple_GET_SIZE(co->co_cellvars) + i] = o;
        }
    }
    ......
}
```

其中的 closure 变量是作为最后一个函数参数传递进来的，我们可以看看在 fast_function 中到底传进来了什么。

```
[funcobject.h]
#define PyFunction_GET_CLOSURE(func)  (((PyFunctionObject *)func) ->
  func_closure)

[ceval.c]
PyObject *fast_function(......)
{
    ......
    return PyEval_EvalCodeEx(......, PyFunction_GET_CLOSURE(func));
}
```

原来传递进来的就是在 PyFunctionObject 对象中与 PyCodeObject 对象绑定的装满了 PyCellObject 对象的 tuple，所以在 PyEval_EvalCodeEx 中，进行的动作就是将这个 PyCellObject 对象一个一个放入到 f_localsplus 中相应的位置。在处理完 closure 之后，inner_func 对应的 PyFrameObject 中的 f_localsplus 如图 11-27 所示。

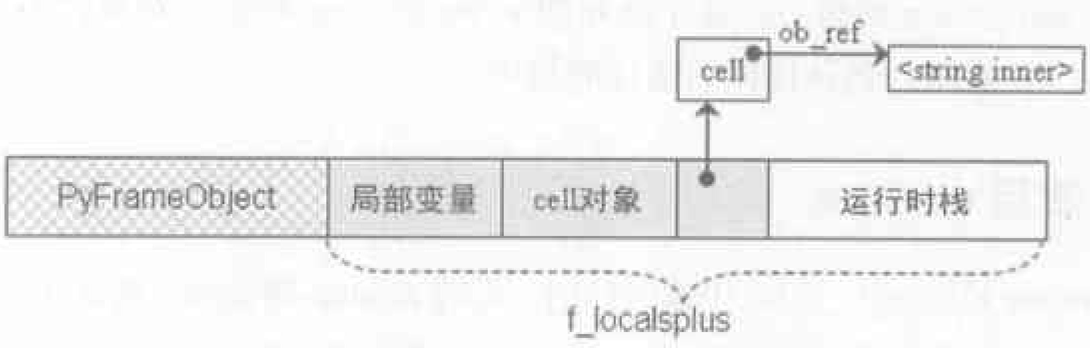



图 11-27 设置 cell 对象之后的 inner_func 函数的 PyFrameObject 对象

这里的动作与调用 get_func 是一致的，所以我们可以猜测，在 inner_func 调用的过程中，当引用外层作用域的符号时，一定是到 f_localsplus 中的 free 变量区域中获得符号对应的值。这正是 inner_func 函数中“print value”表达式对应的第一条字节

码指令"0 LOAD_DEREF 0"的意义。

```
[LOAD_DEREF]
            x = freevars[oparg]; //获得 PyCellObject 对象
            w = PyCell_Get(x);    //获得 PyCellObject.ob_obj 指向的对象
            if (w != NULL) {
                PUSH(w);
                continue;
            }
            ......
```

到现在，我们已经剖析完了 closure 从创建、传递到使用的全过程，再回顾一下图 11-25、图 11-26 和图 11-27，Python 中的 closure 定然能够了然于胸了。

### 11.6.3 Decorator

在 closure 技术的基础上，Python 实现了 decorator。考虑下面的例子：

```
[decorator.py]
def should_say(fn):
    def say(*args):
       print 'say something...'
       fn(*args)
    return say

@should_say
def func():
    print 'in func'

//输出结果为:
//say something…
//in func
func()
```

实际上我们可以完全不用 decorator，而实现相同的效果，只需要对 func 进行小小的修改：

```
[decorator1.py]
......
def func():
    print 'in func'
func=should_say(func)
func()
```

执行 decorator1.py，会发现输出的结果完全相同，同时，基于上面对 closure 的剖析，decorator1.py 的行为也非常好理解了。这个现象提示我们，decorator 似乎就是"func=should_say(func)"的一种包装形式，为了验证这一点，我们来看看两者编译之后的结果：

```
[decorator.py]
......
@should_say
def func():
9   LOAD_NAME 0 (should_say)
12  LOAD_CONST 1 (<code object func>)
15  MAKE_FUNCTION 0
18  CALL_FUNCTION 1
21  STORE_NAME 1 (func)
     print 'in func'
......
```

```
[decorator1.py]
......
def func():
9   LOAD_CONST 1 (<code object func>)
12  MAKE_FUNCTION 0
15  STORE_NAME 1 (func)
     print 'in func'
func=should_say(func)
18  LOAD_NAME 0 (should_say)
21  LOAD_NAME 1 (func)
24  CALL_FUNCTION 1
27  STORE_NAME 1 (func)
......
```

在 decorator1.py 中，“15 STORE_NAME 1”和“21 LOAD_NAME 1”这两条字节码指令互为逆运算，可以删除，如此一来，decorator1.py 编译后的字节码指令序列和 decorator.py 编译后的字节码指令序列除了“LOAD_NAME 0”的位置不同，其余的都完全相同。

很显然，在 Python 中，decorator 仅仅是 decorator1.py 中“func=should_say(func)”的一种包装方式，而理解 decorator 的关键，就在于理解 Python 中的 closure 了。

CHAPTER

第 12 章

# Python 虚拟机中的类机制

本章将研究类机制在 Python 中的实现。从 Python 2.2 开始，Python 中有了两套类机制，一套称为 classic class，而另一套称为 new style class。虽然从使用上来看，两者并没有太大的差别，但是在实现上，两者有很大的区别。随着 Python 的不断演进，classic class 最终将在 Python 中消失，所以在本章中，仅仅考察 Python 中 new style class 机制的实现。

## 12.1 Python 中的对象模型

在 Python 2.2 之前，Python 中存在着一个巨大的裂缝，就是 Python 的内置 `type`，比如 `int`，`dict`，与 Python 程序员定义的 `class` 并不是完全等同的。举一个例子，用户定义的 `class A` 可以被继承，作为另一个 `class B` 的基类；但不幸的是，Python 的内置 `type` 却不能被继承，也就是说你没有办法以类似于 C++、Java 中的继承方式那样，很自然地创建一个继承自 `dict` 的类 `MyDict`。Python 的开发者们在 Python 2.2 中花费了巨大的精力填补了内置 `type` 和用户自定义 class 之间的鸿沟，使这两者能够在概念上实现了完全一致。这种统一后的类型机制，称之为 new style class 机制。

在面向对象的理论中，有两个核心的概念：类和对象。Python 中也实现了这两个概念。但不幸的是，在 Python 中所有的东西都是对象，所以类也是一种对象。对于尝试用文字来描述这种微妙区别的我来说，这真是一场灾难。当我在类、类对象、对象之间不断切换时，用不了多久，不论是我，还是读者，都会发现自己的脑袋像浆糊一样了。所以，我们需要定义一些术语，尝试用另一套结构对 Python 中的类机制建模。

在 Python 2.2 之前，Python 中实际上存在三类对象：

- type 对象：表示 Python 内置的类型；
- class 对象：表示 Python 程序员定义的类型；
- instance 对象（实例对象）：表示由 class 对象创建的实例。

而在 Python 2.2 之后，type 和 class 已经统一，所以我们用"class 对象"来统一地表示 Python 2.2 之前的"type 对象"和"class 对象"。并且我们采用一种表达形式，对于 class 对象 A，我们采用 <class A> 来表示名为 A 的 class 对象；而对于 instance 对象，则采用<instance a>来表示名为 a 的 instance 对象。

同时，我们将采用术语 type 来表示"类型"（注意，不是类型对象）这个概念。比如对于"实例对象 a 的类型是 A"这样的说法，我们就可用"实例对象 a 的 type 是 A"来表达。当然，术语 class 在某种情况下也表示"类型"，比如我们会采用"定义了一个名为 A 的 class"或"class A"这样的说法。但是当我们使用"class 对象"时，就与"class"有完全不同的意义了。"class"表示"类"或"类型"这个概念，而"class 对象"表示这个概念在 Python 中的实现。

在 class 对象与 class 对象之间，class 对象与 instance 对象之间，存在着多种联系。我们将这些对象和它们之间的联系称为类型系统或对象模型。在本书的开始，我们简单地介绍了一下 Python 对象模型，主要关注了 Python 对象在 C 一级的组织。而本章将深入细致地剖析 Python 对象模型的方方面面。

## 12.1.1 对象间的关系

在 Python 的三种对象之间，存在着两种关系：

- is-kind-of 关系：这种关系对应于面向对象中的基类与子类之间的关系；
- is-instance-of 关系：这种关系对应于面向对象中类与实例之间的关系。

考虑下面的 Python 代码：

```
class A(object):
    pass

a = A()
```

其中包含了三个对象：object（class 对象），A（class 对象）和 a（instance 对象）。有面向对象基础的读者一定马上能够指出，object 和 A 之间存在 is-kind-of 关系，即 A 是 object 的子类；而 a 和 A 之间存在 is-instance-of 关系，即 a 是 A 的一个实例。再稍微进行一下推理，显然，a 和 object 之间也存在 is-instance-of 关系，即 a 也是 object 的一个实例。

Python 提供了一些方法可以用来探测这些关系，通过对象的__class__属性或 Python 内置的 type 方法可以探测一个对象和哪个对象存在 is-instance-of 关系；而通过对象的__bases__属性则可探测一个对象和哪个对象存在 is-kind-of 关系。此外，Python 还提供两个内置方法 issubclass 和 isinstanceof 来判断两个对象间是否存在我们期望的关系，图 12-1 显示了利用这些方法探测和验证关系的例子：

```
>>> a.__class__
<class '__main__.A'>
>>> type(a)
<class '__main__.A'>
>>> A.__class__
<type 'type'>
>>> type(A)
<type 'type'>
>>> object.__class__
<type 'type'>
>>> type(object)
<type 'type'>
>>> A.__bases__
(<type 'object'>,)
>>> object.__bases__
()
>>> a.__bases__

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    a.__bases__
AttributeError: 'A' object has no attribute '__bases__'
>>> isinstance(a, A)
True
>>> issubclass(A, object)
True
```

图 12-1 探测对象间的关系

从 a.__bases__的结果看出，并不是所有的对象都拥有 is-kind-of 关系。这也很合理，因为 is-kind-of 关系对应的是基类和子类的关系，显然只能在 class 对象与 class 对象之间存在，而 a 是一个 instance 对象，显然不能拥有这种关系。

图 12-2 则更加形象和清晰地展示了这三个对象之间的关系：

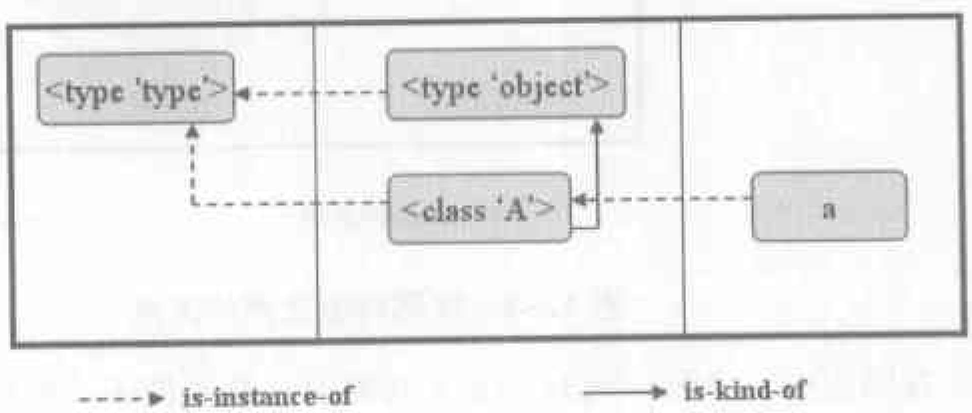



图 12-2 对象关系图

为什么要将两种对象分为三列呢？因为最左边的<type 'type'>这个对象非常特殊，马上我们就会看到。

## 12.1.2 <type 'type'>和<type 'object'>

将 object 和 A 放在一起是因为它们有一个共同的特点，即它们的 type 都是<type 'type'>。<type'type'>属于 Python 中的一种特殊的 class 对象，这种特殊的 class 对象能够成为其他 class 对象的 type。这种特殊的 class 对象我们称之为 metaclass 对象，在本章中，只涉及<type 'type'>这一个 metaclass 对象，而一般的 class 对象，仍以“class 对象”称之。在 Python 中，metaclass 对象的意义非常重大，在后面的剖析中我们可以看到，创建一个 class 对象的关键之处就在于 metaclass 对象。

Python 中还有一个特殊的 class 对象——<type 'object'>，在 Python 中，任何一个 class 都必须直接或间接继承自 object，这个 object 可以视为万物之母。

在<type 'type'>和<type 'object'>之间有非常微妙的关系，通过图 12-3 可以看到这种微妙的关系，同时，图 12-3 还探测了其他一些 type 对象与<type 'type'>和<type 'object'>之间的关系。

```
>>> object.__class__
<type 'type'>
>>> type.__class__
<type 'type'>
>>> type.__bases__
(<type 'object'>,)
>>> int.__class__
<type 'type'>
>>> int.__bases__
(<type 'object'>,)
>>> dict.__class__
<type 'type'>
>>> dict.__bases__
(<type 'object'>,)
```

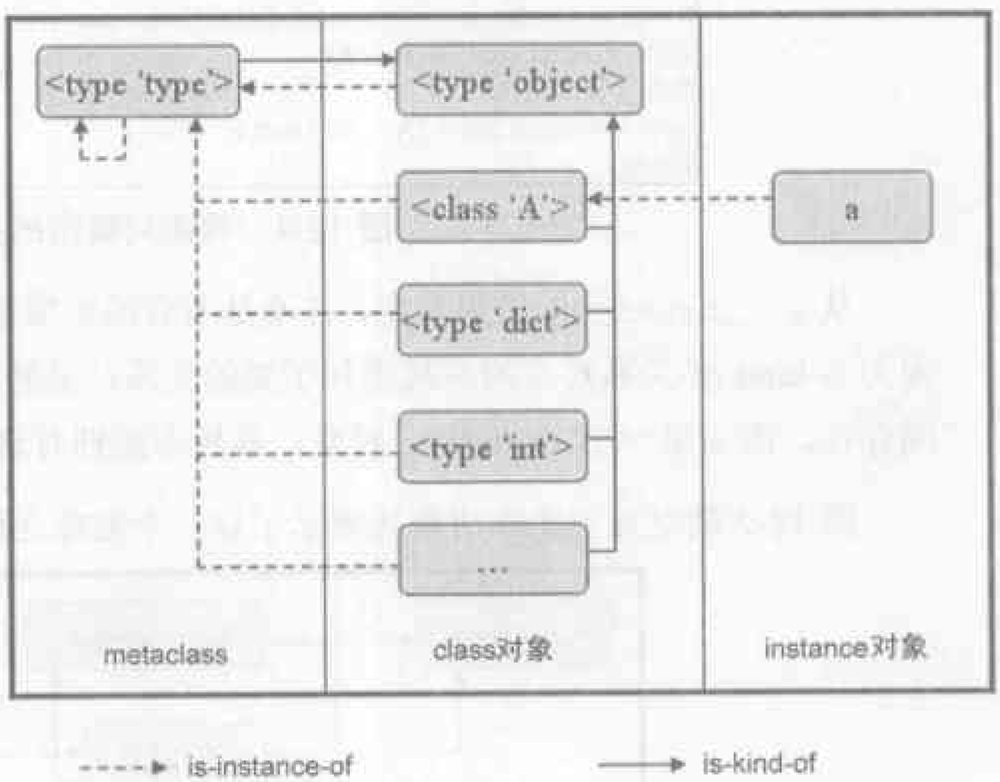



图 12-3 探测对象之间的关系

图 12-3 右侧图中，中间一列的 class 对象有一种类似于“波粒二相性”的特殊性质，我们说 Python 中的对象分为 class 对象和 instance 对象，但中间这一列的对象既是 class 对象，又是 instance 对象。说它是 class 对象，因为它可以通过实例化的动作创

建新的 instance 对象；说它是 instance 对象，因为它确实是 metaclass 对象经过实例化得到的，图 12-3 也显示出 class 对象和 metaclass 对象之间存在 is-instance-of 关系。这种二相性对于理解后面的内容有着非常重要的意义。

现在我们可以总结一下：

- 在 Python 中，任何一个对象都有一个 type，可以通过对象的__class__属性获得。任何一个 instance 对象的 type 都是一个 class 对象，而任何一个 class 对象的 type 都是 metaclass 对象。在大多数情况下这个 metaclass 都是<type 'type'>，而在 Python 内部，它实际上对应的就是 PyType_Type。
- 在 Python 中，任何一个 class 对象都直接或间接与<type 'object'>对象之间存在 is-kind-of 关系，包括<type 'type'>。在 Python 内部，<type 'object'>对应的是 PyBaseObject_Type。

## 12.2 从 type 对象到 class 对象

在 Python 中，实现“类”这个概念的是 class 对象，这个对象听上去神秘，实际上，我们之前对这个对象早已司空见惯了。在 Python 内部，class 对象其实就是一个 PyObject 结构体。那么我们之前看到过的 PyInt_Type，PyList_Type，这些都是 class 对象了？是，也不是。呃，至少目前还不是。目前我们对诸如 PyInt_Type，PyList_Type 的认识还停留在 Python 2.2 之前，前面说了，这时的 PyInt_Type 应该叫做 type 对象，跟 Python 2.2 之后的 class 对象还存在着差别。前面提到在 Python 2.2 之前不能继承内置类型，那好，我们来尝试一下：

```
class MyInt(int):
    def __add__(self, other):
        return int.__add__(self, other) + 10

a = MyInt(1)
b = MyInt(2)
print a + b
```

从内置的 int 类型继承得到一个新的整数类型，这种类型的整数在进行加法操作时，会在正常加法结果的基础上再加上 10。现在用我们的大脑模拟一下 Python 虚拟机，当 a + b 发生时，会调用 MyInt.__add__，在这里，调用了 int.__add__。问题从这里现身了，Python 中的 int 对应着整数对象的类型对象，即 PyInt_Type，其中倒是有一个 nb_add (int_add) 可以完成加法操作，但是，Python 虚拟机该怎么从 int.__add__得知要调用的是 PyInt_Type.tp_as_number.nb_add 呢？这就是为什么 Python 2.2 之前的内置类型不能被继承的原因。因为没有在 type 中寻找某个属性的机制。

这里先以 <type 'int'>（也就是 PyInt_Type）为例，在图 12-4 中先给出 Python 中 class 对象的一个粗略的图示，从中可以看到 Python 2.2 之后是如何解决这个属性寻找机制的，在此后的剖析中，我们的工作就是不断将这个粗略的图示完善。

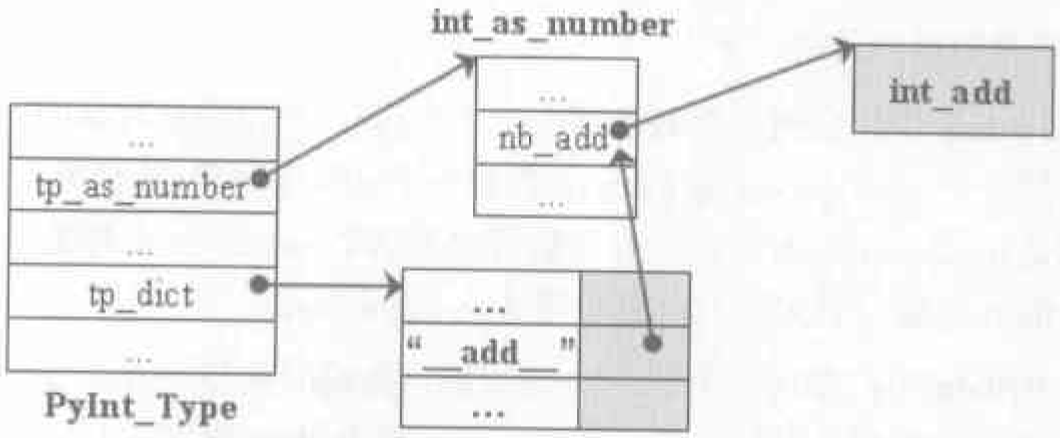



图 12-4 粗略的<type 'int'>示意图

当 Python 虚拟机需要调用 int.__add__时，它可以到符号"int"对应的 class 对象——PyInt_Type——的 tp_dict 指向的 dict 对象中查找符号"__add__"对应的操作，并调用该操作，从而完成对 int.__add__的调用。

图 12-4 仅仅是一个粗略的示意图，实际上在 tp_dict 中与符号"__add__"对应的对象虽然与 nb_add 有关系，但并不是直接指向 nb_add 的，我们说可以调用__add__对应的操作，那么这个对象是不是就是我们 11 章看到的 PyFunctionObject 对象呢，毕竟，"函数调用"听上去才合理呀。但实际上，在 Python 中，不仅只有函数可以被调用，一切对象都有可能被调用。

到了这里，需要特别提出一个 Python 中的概念，即**可调用性**（**callable**），只要一个对象对应的 class 对象中实现了"__call__"操作（更确切地说，在 Python 内部的 PyTypeObject 中，tp_call 不为空）那么这个对象就是一个可调用的对象，换句话说，在 Python 中，**所谓"调用"，就是执行对象的 type 所对应的 class 对象的 tp_call 操作**。图 12-5 展示了一个可调用的对象：

```
>>> class A(object):
        def __call__(self):
            print 'Hello Python'

>>> a = A()
>>> a()
Hello Python
```

图 12-5 可调用对象示例

实际上，熟悉 C++的朋友可以将这个特性看作是 C++中通过对操作符()的重载实现 Functor 的技术。在 Python 内部，是通过一个名为 PyObject_Call 的函数对 instance 对象 a 进行操作，从而调用 a 中的__call__，完成"可调用"这个特性的。

图 12-6 中展示了一个不可调用的例子，从这里可以看到，PyObject_Call 对于任何对象都会按部就班地试图完成“调用”这个操作，如果传递进来的对象确实是可调用的，那么调用操作自然能够顺利完成；如果传递进来的对象本身并不是可调用的，那么很显然，Python 将会抛出异常：

```
>>> def f():
        i = 1
        i()


>>> f()

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    f()
  File "<pyshell#9>", line 3, in f
    i()
TypeError: 'int' object is not callable
```

图 12-6　不可调用对象示例

看，一个整数对象是不可调用的。有趣的是，一个对象是否可调用并不是在编译期能确定的，必须是在运行时才能在 PyObject_CallFunctionObjArgs 中确定，所以在图 12-6 中你明知道 a = i()这条语句一定失败，而 Python 却成功编译了。

在目前的 Python 官方网站 *www.python.org* 上，我们还能下载到类机制革命发生前后的两个 Python 发行版本，Python 2.1.3 和 Python 2.2.3。对比这两个版本的 object.h 文件，会发现 PyTypeObject 的定义有很多不同，tp_dict 在 Python 2.1.3 中不存在，在 Python 2.2.3 中才开始冒头。这个 tp_dict 也正是 PyTypeObject 从 2.2 之前的 type 对象向 2.2 之后的 class 对象转变的关键。虽然有了 tp_dict 这个域，而且我们已经在图 12-4 中看到，tp_dict 在运行时会指向一个 dict 对象。这个 dict 对象必须在运行时动态构建。

从 Python 2.2 开始，Python 在启动时，会对类型系统（对象模型）进行初始化的动作。这个初始化的动作会动态地在内置类型对应的 PyTypeObject 中填充一些重要的东西，其中当然也包括填充 tp_dict，从而完成内置类型从 type 对象到 class 对象的转变。这个对类型系统进行初始化的动作从_Py_ReadyTypes 拉开序幕。

在_Py_ReadyTypes 中，会调用 PyType_Ready 对 class 对象进行初始化。实际上，PyType_Ready 仅仅是属于对 class 对象进行初始化这个动作的一部分，它处理的不光是 Python 的内置类型，还会处理用户定义的类型。我们以 list 和 A 来说明内置类型与用户自定义类型在初始化上的区别。list 对应的 class 对象 PyList_Type 在 Python 启动后已经作为全局对象存在了，需要的仅仅是完善；而 A 对应的 class 对象则并不存在，需要申请内存，并创建、初始化整个动作序列。所以对于 list 来说，初始化就剩下 PyType_Ready

了，而对于 A 来说，PyType_Ready 仅仅是很小的一部分。

既然之前提到<type 'type'>是一个非常特殊的 class 对象，那么对 PyType_Ready 的考察就以它作为参数吧。<type 'type'>实际上对应着 PyType_Type，所以对 PyType_Ready 的剖析就从 PyType_Ready(&PyType_Type)开始。

## 12.2.1 处理基类和 type 信息

**代码清单 12-1**

```
[typeobject.c]
int PyType_Ready(PyTypeObject *type)
{
  PyObject *dict, *bases;
  PyTypeObject *base;
  Py_ssize_t i, n;

  //[1]: 尝试获得 type 的 tp_base 中指定基类(super type)
  base = type->tp_base;
  if (base == NULL && type != &PyBaseObject_Type) {
      base = type->tp_base = &PyBaseObject_Type;
  }

  //[2]: 如果基类没有初始化，先初始化基类
  if (base && base->tp_dict == NULL) {
      PyType_Ready(base)
  }

  //[3]: 设置 type 信息
  if (type->ob_type == NULL && base != NULL)
      type->ob_type = base->ob_type;
  ......
}
```

首先在代码清单 12-1 中的[1]处，Python 虚拟机会尝试获得待初始化的 type 的基类(注意，这里的 type 是 PyType_Ready 中的参数名，也表示其对应的 class 对象)。这个信息是在 PyTypeObject.tp_base 中指定的。表 12-1 列出了一些内置 class 对象的 tp_base 信息：

表 12-1 内置 class 对象的基类信息

| class 对象 | 基类信息 |
|---|---|
| PyType_Type | NULL |
| PyInt_Type | NULL |
| PyBool_Type | &PyInt_Type |

对于指定了 tp_base 的内置 class 对象，当然就使用指定的基类；而对于没有指定

tp_base 的内置 class 对象，Python 将为其指定一个默认的基类：PyBaseObject_Type。前面说了，这个东西就是那个特殊的<type 'object'>。所以这里可以看到，Python 所有 class 对象都是直接或间接以<type 'object'>作为基类的。我们正在考察的 PyType_Type 很倒霉，它没有指定基类，所以它的基类就成了<type 'object'>。

在获得了基类后，代码清单 12-1 的[2]处会判断基类是否已经被初始化了，如果没有，则需要先对基类进行初始化。可以看到，判断初始化是否完成的条件是 base->tp_dict 是否为 NULL，这符合之前对初始化的描述，初始化的一部分工作就是对 tp_dict 进行填充。

随后在代码清单 12-1 的[3]处，设置了 class 对象的 ob_type 信息，实际上这个 ob_type 信息也就是对象的__class__将返回的信息。更进一步地说，这里设置的 ob_type 就是 metaclass。实际上，Python 虚拟机是将基类的 metaclass 作为了子类的 metaclass。对于这里考察的 PyType_Type 来说，其 metaclass 正是<type 'object'>的 metaclass，而在 PyBaseObject_Type 的定义中我们可以看到其 ob_type 被设置成了 PyType_Type，所以嘛……现在能理解图 12-3 的结果了吧☺。

既然在代码清单 12-1 的[2]处我们看到 Python 虚拟机会首先尝试初始化 PyBaseObject_Type，同时，<type 'object'>又是所有 class 对象的基类，那么我们转而分析 PyType_Ready (&PyBaseObject_Type)吧。对于 PyBaseObject_Type，代码清单 12-1 的[1]、[2]、[3]处的动作读者可以自行分析，其实结果很简单，就是什么动作也没有发生。

## 12.2.2 处理基类列表

接下来，Python 虚拟机将处理类型的基类列表，因为 Python 支持多重继承，所以每一个 Python 的 class 对象都会有一个基类列表。

```
[typeobject.c]
int PyType_Ready(PyTypeObject *type)
{
    PyObject *dict, *bases;
    PyTypeObject *base;
    Py_ssize_t i, n;
    ……
    //尝试获得 type 的 tp_base 中指定基类(super type)
   base = type->tp_base;
   if (base == NULL && type != &PyBaseObject_Type) {
       base = type->tp_base = &PyBaseObject_Type;
   }
    ……
    //处理 bases ：基类列表
    bases = type->tp_bases;
    if (bases == NULL) {
        //如果 bases 为空，则根据 base 的情况设定 bases
        if (base == NULL)
```

```
            bases = PyTuple_New(0);
        else
            bases = PyTuple_Pack(1, base);
        type->tp_bases = bases;
    }
}
```

对于我们现在考察的 PyBaseObject_Type 来说，其 tp_bases 为空，而其 base 也为 NULL，所以它的基类列表就是一个空的 tuple 对象。这也符合在图 12-1 中显示的 object.__bases__ 的结果。

而对于 PyType_Type 和其他类型，比如 PyInt_Type 来说，虽然 tp_bases 为空，但是 base 不为 NULL，而是&PyBaseObject_Type，所以它们的基类列表不为空，都包含一个 PyBaseObject_Type，这也可以从图 12-1 和图 12-3 中得到验证。

### 12.2.3 填充 tp_dict

接下来 Python 虚拟机将进入激动人心的填充 tp_dict 的阶段，这是一个极其繁杂的过程。

```
[typeobject.c]
int PyType_Ready(PyTypeObject *type)
{
    PyObject *dict, *bases;
    PyTypeObject *base;
    Py_ssize_t i, n;
    ……
    //设定 tp_dict
    dict = type->tp_dict;
    if (dict == NULL) {
        dict = PyDict_New();
        type->tp_dict = dict;
    }

    //将与 type 相关的 descriptor 加入到 tp_dict 中
    add_operators(type);
    if (type->tp_methods != NULL) {
        add_methods(type, type->tp_methods);
    }
    if (type->tp_members != NULL) {
        add_members(type, type->tp_members);
    }
    if (type->tp_getset != NULL) {
        add_getset(type, type->tp_getset);
    }
    ……
}
```

在这个阶段，完成了将（“__add__”，&nb_add）加入到 tp_dict 的过程。这个阶段的 add_operators、add_methods、add_members、add_getset 都是完成这样的填充

tp_dict 的动作。那么，一个问题浮现了，Python 虚拟机是如何知道“__add__”和 nb_add 之间存在关联的呢？这种关联是在 Python 源代码中预先就确定好了的，存放在一个名为 slotdefs 的全局数组中。

#### 12.2.3.1 slot 与操作排序

在进入填充 tp_dict 的复杂操作之前，我们先来介绍 Python 内部的一个概念：slot。在 Python 内部，slot 可以视为表示 PyTypeObject 中定义的操作，在一个操作对应一个 slot，但是 slot 又不仅仅包含一个函数指针，它还包含其他一些信息。在 Python 内部，slot 是通过 slotdef 这个结构体来实现的。

```
[typeobject.c]
typedef struct wrapperbase slotdef;

[descrobject.h]
struct wrapperbase {
    char *name;
    int offset;
    void *function;
    wrapperfunc wrapper;
    char *doc;
    int flags;
    PyObject *name_strobj;
};
```

在一个 slot 中，存储着与 PyTypeObject 中一种操作相对应的各种信息。比如，name 就是操作对应的名称，比如字符串“__add__”；offset 则是操作的函数地址在 PyHeapTypeObject 中的偏移量；而 function 则指向一种称为 slot function 的函数。

Python 中提供了多个宏来定义一个 slot，其中最基本的是 TPSLOT 和 ETSLOT：

```
[typeobject.c]
#define TPSLOT(NAME, SLOT, FUNCTION, WRAPPER, DOC) \
    {NAME, offsetof(PyTypeObject, SLOT), (void *)(FUNCTION), WRAPPER, \
    PyDoc_STR(DOC)}
#define ETSLOT(NAME, SLOT, FUNCTION, WRAPPER, DOC) \
    {NAME, offsetof(PyHeapTypeObject, SLOT), (void *)(FUNCTION), WRAPPER, \
    PyDoc_STR(DOC)}

[structmember.h]
#define offsetof(type, member) ( (int) & ((type*)0) -> member )
```

TPSLOT 和 ETSLOT 的区别在于 TPSLOT 计算的是操作对应的函数指针（比如 nb_add）在 PyTypeObject 中的偏移量，而 ETSLOT 计算的是函数指针在 PyHeapTypeObject 中的偏移量。但是观察下面列出的 PyHeapTypeObject 的代码，可以发现，因为 PyHeapTypeObject 的第一个域就是 PyTypeObject，所以 TPSLOT 计算出的偏移量实际上也就是相对于 PyHeapTypeObject 的偏移量。

对于一个 PyTypeObject 来说，有的操作，比如 nb_add，其函数指针是在 PyNumberMethods 中存放的，而 PyTypeObject 中却是通过一个 tp_as_number 指针指向另一个 PyNumberMethods 结构，所以，实际上根本没有办法计算出 nb_add 在 PyTypeObject 中的偏移量，只能计算其在 PyHeapTypeObject 中的偏移量。

```
[object.h]
typedef struct _heaptypeobject {
    PyTypeObject ht_type;
    PyNumberMethods as_number;
    PyMappingMethods as_mapping;
    PySequenceMethods as_sequence;
    PyBufferProcs as_buffer;
    PyObject *ht_name, *ht_slots;
} PyHeapTypeObject;
```

因此，与 nb_add 对应的 slot 必须是通过 ETSLOT 来定义的。到了这里，细心的读者一定发现了一个重大的问题，如果说与 nb_add 对应的 slot 中记录的 offset 是基于 PyHeadTypeObject 的，而 PyInt_Type 却是一个 PyTypeObject，那么显然通过这个偏移量不可能得到 PyInt_Type 中为 int 准备的 nb_add。呃，那要这个劳什子的 offset 有什么用呢？

答案非常诡异，真的，这个 offset 是用来对操作进行排序的。排序？排哪门子的序呢？别急，为了理解为什么需要对操作进行排序，需要来看看 Python 预先定义的 slot 集合——slotdefs。

```
[typeobject.c]
......
#define SQSLOT(NAME, SLOT, FUNCTION, WRAPPER, DOC) \
    ETSLOT(NAME, as_sequence.SLOT, FUNCTION, WRAPPER, DOC)
......

static slotdef slotdefs[] = {
    ......
    //[不同操作名对应相同操作]
    BINSLOT("__add__", nb_add, slot_nb_add, "+"),
    RBINSLOT("__radd__", nb_add, slot_nb_add, "+"),
    //[相同操作名对应不同操作]
    SQSLOT("__getitem__", sq_item, slot_sq_item, wrap_sq_item,
           "x.__getitem__(y) <==> x[y]"),
    MPSLOT("__getitem__", mp_subscript, slot_mp_subscript, wrap_binaryfunc,
           "x.__getitem__(y) <==> x[y]"),
    ......
}
```

其中的 BINSLOT，SQSLOT 等这些宏实际上都是对 ETSLOT 的一个简单包装。在 slotdefs 中，可以发现，操作名（比如__add__）和操作并不是一一对应的，存在着多个操作对应同一个操作名的情况，同样也存在着同一个操作对应不同操作名的情况。对于相同操作名对应不同操作的情况，在填充 tp_dict 时，就会出现问题，比如对于

"__getitem__"，在 tp_dict 中与其对应的是 sq_item，还是 mp_subscript 呢？

为了解决这个问题，就需要利用 slot 中的 offset 信息对 slot（也就是对操作）进行排序。回顾一下前面列出的 PyHeapTypeObject 的代码，它与一般的 struct 定义不同，其实定义中各个域的顺序是相当关键的，在顺序中隐含着操作优先级的信息。比如在 PyHeapTypeObject 中，PyMappingMethods 的位置在 PySequenceMethods 之前，mp_subscript 是 PyMappingMethods 中的 PyObject*，而 sq_item 是 PySequenceMethods 中的 PyObject*，所以最终计算出的偏移存在如下的关系：offset(mp_subscript) < offset(sq_item)。如果一个 PyTypeObject，既定义了 mp_subscript，又定义了 sq_item，那么 Python 虚拟机将选择 mp_subscript 与"__getitem__"建立联系。非常幸运，PyList_Type 正是这样的一个 PyTypeObject，在 PyList_Type 中，tp_as_mapping.mp_subscript 指向 list_subscript，而 tp_as_sequence.sq_item 指向 list_item，我们可以在两者中输出信息，来查看在这场宫廷选秀中，究竟花落谁家。结果如图 12-7 所示：

```
>>> class A(list):
        pass

>>> a = A()
>>> a.append(1)
>>> print a[0]
call list_subscript
call list_item
1
```

图 12-7 探测 list 中__getitem__对应的操作

因为 Python 对 list 的索引元素的操作有优化，所以我们必须从 list 派生出一个自定义类，才能看出结果，A 中的"__getitem__"对应的操作就是对 PyList_Type 中的 mp_subscript 和 sq_item 选择的结果。可以看到确实 list_subscript 被选中了，至于为什么之后还会输出一个调用 list_item 的信息☺，答案非常简单，也非常诡异，因为在 list_subscript 中调用了函数 list_item。

整个对 slotdefs 的排序在 init_slotdefs 中完成：

```
[typeobject.c]
static void init_slotdefs(void)
{
    slotdef *p;
    static int initialized = 0;
    //init_slotdefs 只会进行一次
    if (initialized)
        return;
    for (p = slotdefs; p->name; p++) {
        //填充 slotdef 结构体中的 name_strobj
        p->name_strobj = PyString_InternFromString(p->name);
    }
```

```
    //对 slotdefs 中的 slotdef 进行排序
    qsort((void *)slotdefs, (size_t)(p-slotdefs), sizeof(slotdef),
        slotdef_cmp);
    initialized = 1;
}

//slot 排序的比较策略
static int slotdef_cmp(const void *aa, const void *bb)
{
    const slotdef *a = (const slotdef *)aa, *b = (const slotdef *)bb;
    int c = a->offset - b->offset;
    if (c != 0)
        return c;
    else
        return (a > b) ? 1 : (a < b) ? -1 : 0;
}
```

在 slot 的排序策略函数 slotdef_cmp 中，可以清晰地看到，slot 中的 offset 正是操作排序的关键所在。

### 12.2.3.2 从 slot 到 descriptor

在 slot 中，包含了很多关于一个操作的信息，但是很可惜，在 tp_dict 中，与"__getitem__"关联在一起的，一定不会是一个 slot。原因很简单，slot 不是一个 PyObject，它不能存放在 dict 对象中。当然，如果我们再深入地思考一下，会发现，slot 也不会被调用。既然 slot 不是一个 PyObject，那么它就没有 type，也就无从谈起什么 tp_call 了，所以 slot 是无论如何也不能满足前面描述的 Python 中的"可调用"这个概念。

前面我们说过，Python 虚拟机在 tp_dict 找到"__getitem__"对应的"操作"后，会调用该"操作"，所以在 tp_dict 中与"__getitem__"对应的只能是另一个包装了 slot 的 PyObject，在 Python 中，这是一个我们称之为 descriptor 的东西。

在 Python 内部，存在多种 descriptor，与 PyTypeObject 中的操作对应的是 PyWrapperDescrObject。在此后的描述中，我们将用术语 descriptor 来专门表示 PyWrapperDescrObject。一个 descriptor 包含一个 slot，其创建是通过 PyDescr_NewWrapper 完成的。

```
[descrobject.h]
#define PyDescr_COMMON \
    PyObject_HEAD \
    PyTypeObject *d_type; \
    PyObject *d_name

typedef struct {
    PyDescr_COMMON;
    struct wrapperbase *d_base;
    void *d_wrapped; /* This can be any function pointer */
} PyWrapperDescrObject;
```

```
[descrobject.c]
static PyDescrObject *
descr_new(PyTypeObject *descrtype, PyTypeObject *type, char *name)
{
    PyDescrObject *descr;
    //申请空间
    descr = (PyDescrObject *)PyType_GenericAlloc(descrtype, 0);
    descr->d_type = type;
    descr->d_name = PyString_InternFromString(name);
    return descr;
}

PyObject *
PyDescr_NewWrapper(PyTypeObject *type, struct wrapperbase *base, void
  *wrapped)
{
    PyWrapperDescrObject *descr;
    descr = descr_new(&PyWrapperDescr_Type, type, base->name);
    descr->d_base = base;
    descr->d_wrapped = wrapped;
    return (PyObject *)descr;
}
```

Python 内部的各种 descriptor 都将包含 PyDescr_COMMON，其中的 d_type 被设置为 PyDescr_NewWrapper 的参数 type，而 d_wrapped 则存放着最重要的信息：操作对应的函数指针，比如对于 PyList_Type 来说，其 tp_dict["__getitem__"].d_wrapped 就是&mp_subscript。而 slot 则被存放在了 d_base 中。

PyWrapperDescrObject 的 type 是 PyWrapperDescr_Type，其中的 tp_call 是 wrapperdescr_call，当 Python 虚拟机"调用"一个 descriptor 时，也就会调用 wrapperdescr_call。对于 descriptor 的"调用"过程，我们将在以后详细剖析。

#### 12.2.3.3 建立联系

排序后的结果仍然存放在 slotdefs 中，Python 虚拟机这下就可以从头到尾遍历 slotdefs，基于每一个 slot 建立一个 descriptor，然后在 tp_dict 中建立从操作名到 descriptor 的关联。这个过程在 add_operators 中完成：

```
[typeobject.c]
static int add_operators(PyTypeObject *type)
{
    PyObject *dict = type->tp_dict;
    slotdef *p;
    PyObject *descr;
    void **ptr;
    //对 slotdefs 进行排序
    init_slotdefs();
    for (p = slotdefs; p->name; p++) {
        //如果 slot 中没有指定 wrapper，则不处理
        if (p->wrapper == NULL)
```

```
            continue;
        //获得 slot 对应的操作在 PyTypeObject 中的函数指针
        ptr = slotptr(type, p->offset);
        //如果 tp_dict 中已经存在操作名，则放弃
        if (PyDict_GetItem(dict, p->name_strobj))
            continue;
        //创建 descriptor
        descr = PyDescr_NewWrapper(type, p, *ptr);
        //将（操作名，descriptor）放入 tp_dict 中
        PyDict_SetItem(dict, p->name_strobj, descr);
    }
    return 0;
}
```

在 add_operators 中，首先会调用前面剖析过的 init_slotdefs 对操作进行排序，然后遍历排序完成后的 slotdefs 结构体数组，对其中的每一个 slot(slotdef)，通过 slotptr 获得该 slot 对应的操作在 PyTypeObject 中的函数指针，并接着创建 descriptor，在 tp_dict 中建立从操作名（slotdef.name_strobj）到操作（descriptor）的关联。

需要注意的是，在创建 descriptor 之前，Python 虚拟机会检查在 tp_dict 中操作名是否已经存在，如果已经存在，则不会再次建立从操作名到操作的关联。正是这种检查机制与上一节的排序机制相结合，使得 Python 虚拟机能够在拥有相同操作名的多个操作中选择优先级最高的操作。

在 add_operators 中，上面描述的动作都很直观、简单。而最难的动作隐藏在 slotptr 这个函数中。它的功能是完成从 slot 到 slot 对应操作的真实函数指针的转换。我们已经知道，在 slot 中存放着操作的 offset，但是很不幸，这个 offset 是相对于 PyHeapTypeObject 的偏移，而操作的真实函数指针则在 PyTypeObject 中指定。更不幸的是，PyTypeObject 和 PyHeapTypeObject 不是同构的，因为 PyHeapTypeObject 中包含了 PyNumberMethods 结构体，而 PyTypeObject 中只包含了 PyNumberMethods*。所以 slot 中存储的这个关于操作的 offset 对于 PyTypeObject 来说，不可能直接使用，必须通过转换。

举个例子，假如说调用 slotptr(&PyList_Type, offset(PyHeapTypeObject, mp_subscript))，首先判断这个偏移大于 offset(PyHeapTypeObject, as_mapping)，所以会先从 PyTypeObject 对象中获得 as_mapping 指针 p，然后在 p 的基础上进行偏移就可以得到实际的函数的地址，而偏移量 delta 为：

```
delta = offset(PyHeapTypeObject, mp_subscripte) - offset(PyHeapTypeObject,
  as_mapping)
```

这个复杂的转换过程就在 slotptr 中完成：

```
[typeobject.c]
static void** slotptr(PyTypeObject *type, int offset)
{
```

```
    char *ptr;

    //判断从 PyHeapTypeObject 中排在后面的 PySequenceMethods 开始
    if (offset >= offsetof(PyHeapTypeObject, as_sequence)) {
        ptr = (void *)type->tp_as_sequence;
        offset -= offsetof(PyHeapTypeObject, as_sequence);
    }
    else if (offset >= offsetof(PyHeapTypeObject, as_mapping)) {
        ptr = (void *)type->tp_as_mapping;
        offset -= offsetof(PyHeapTypeObject, as_mapping);
    }
    else if (offset >= offsetof(PyHeapTypeObject, as_number)) {
        ptr = (void *)type->tp_as_number;
        offset -= offsetof(PyHeapTypeObject, as_number);
    }
    else {
        ptr = (void *)type;
    }
    if (ptr != NULL)
        ptr += offset;
    return (void **)ptr;
}
```

为什么判断必须首先从 PyHeapTypeObject 中排在后面的 PySequenceMethods 开始，然后向前，依次判断 PyMappingMethods 和 PyNumberMethods 呢？假设我们首先从 PyNumberMethods 开始判断，如果一个操作的 offset 大于在 PyHeapTypeObject 中，as_number 在 PyNumberMethods 的偏移量，那么我们还是没有办法确认这个操作是属于 PyNumberMethods 的？还是属于 PyMappingMethods 或 PySequenceMethods 的？只有从后向前进行判断，才能解决这个问题。

好了，现在我们终于能够摸清楚 Python 在改造 PyTypeObject 时对 tp_dict 做了什么手脚了。图 12-8 显示了 PyList_Type 完成初始化之后的整个布局。其中包括了我们这里讨论的 descriptor 和 slot：

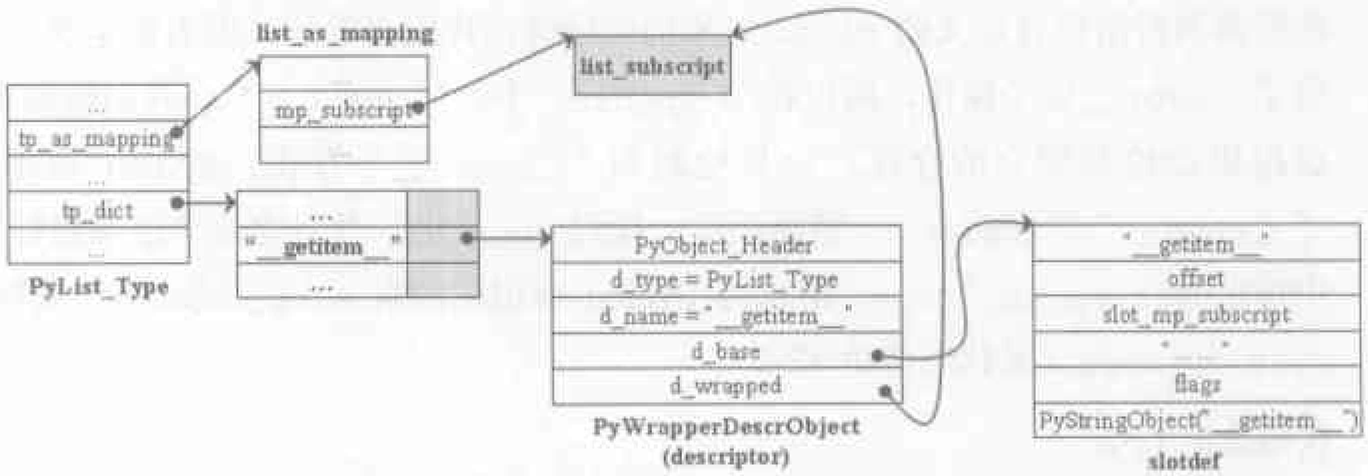



图 12-8 add_operators 完成后的 PyList_Type

在图 12-8 中，从 PyList_Type.tp_as_mapping 中延伸除去的部分是在编译时就已经确定好了的，而从 tp_dict 中延伸出去的部分是在 Python 运行时环境初始化时才建立的。

PyType_Ready 在通过 add_operators 添加了 PyTypeObject 对象中定义的一些 operator 后，还会通过 add_methods、add_members 和 add_getsets 添加在 PyTypeObject 中定义的 tp_methods、tp_members 和 tp_getset 函数集。这些 add_***的过程与 add_operators 类似，不过最后添加到 tp_dict 中的 descriptor 就不再是 PyWrapperDescrObject，而分别是 PyMethodDescrObject、PyMemberDescrObject、PyGetSetDescrObject。

图 12-8 所显示的 class 对象大部分都正确了，但是可惜，它还不是完全正确。考虑图 12-9 显示的例子：

```
>>> class A(list):
        def __repr__(self):
            return 'Python'


>>> s = '%s' % A()
>>> s
'Python'
>>>
```

图 12-9 覆盖 list 特殊操作的类

熟悉 Python 的读者都知道，__repr__是一个 Python 中的 special method。当 Python 虚拟机执行表达式“s = '%s' % A()”时，会最终调用 A.tp_repr，如果按照图 12-8 所示的布局，并且对照 PyList_Type，那么就应该调用 list_repr 这个函数，但并不是这样的，Python 虚拟机最终调用的是我们在 A 中重写之后的 __repr__。这意味着 Python 在初始化 A 时，对 tp_repr 进行了特殊处理。为什么 Python 虚拟机会知道要对 tp_repr 进行特殊处理呢？答案还是在 slot 身上。

在 slotdefs 中，有一条 slot 为 TPSLOT("__repr__", tp_repr, slot_tp_repr, ……)，Python 虚拟机在初始化 A 时，会检查 <class A> 的 tp_dict 中是否存在“__repr__”。在后面剖析用户自定义的 class 对象的创建时，我们会看到，因为在定义 class A 时重写了__repr__这个操作，所以在 A.tp_dict 中，“__repr__”一开始就会存在，Python 虚拟机会检测到它的存在。一旦检测到“__repr__”存在，Python 虚拟机就会根据“__repr__”对应的 slot 顺藤摸瓜，找到 tp_repr，并且将这个函数指针替换为 slot 中指定的&slot_tp_ repr。所以当 Python 虚拟机后来 A.tp_repr 时，实际上执行的是 slot_tp_repr（见代码清单 12-2）。

代码清单 12-2

```
[typeobject.c]
static PyObject* slot_tp_repr(PyObject *self)
{
    PyObject *func, *res;
    static PyObject *repr_str;
    //[1] : 查找"__repr__"属性
```

```
    func = lookup_method(self, "__repr__", &repr_str);
    //[2] : 调用"__repr__"对应的对象
    res = PyEval_CallObject(func, NULL);
}
```

在 slot_tp_repr 中，会寻找"__repr__"属性对应的对象，正好就会找到我们在 A 的定义中重写的函数，后面会看到，这个对象实际上是一个 PyFunctionObject。这样一来，就完成了对默认的 list 的 repr 行为的替换。所以对于 A 来说，其初始化结束后的内存布局则会如图 12-10 所示：

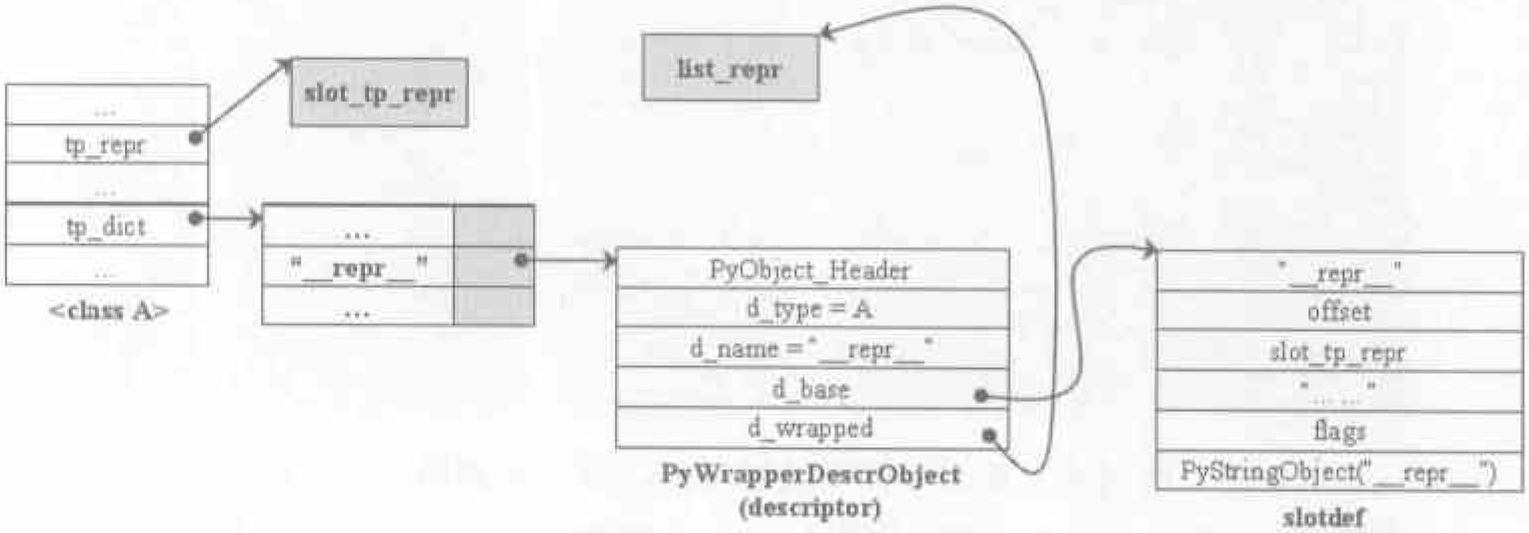



图 12-10 初始化完成后的 A

当然，并不是 A 中所有的操作都会有这样的变化。A 中其他操作还是会指向 PyList_Type 中指定的函数，比如 tp_iter 还是会指向 list_iter。

对于 A 来说，这个变化是在 fixup_slot_dispatchers(PyTypeObject* type)中完成的，对于内置 class 对象，不会进行这个操作。这个操作实际上是属于创建自定义 class 对象时的动作，这里为了完整地介绍初始化结束后的 class 对象的布局，特提前在此叙述。图 12-11 显示了在 Visual Studio 中观察到的 fixup_slot_dispatchers 前后 tp_repr 的变化：

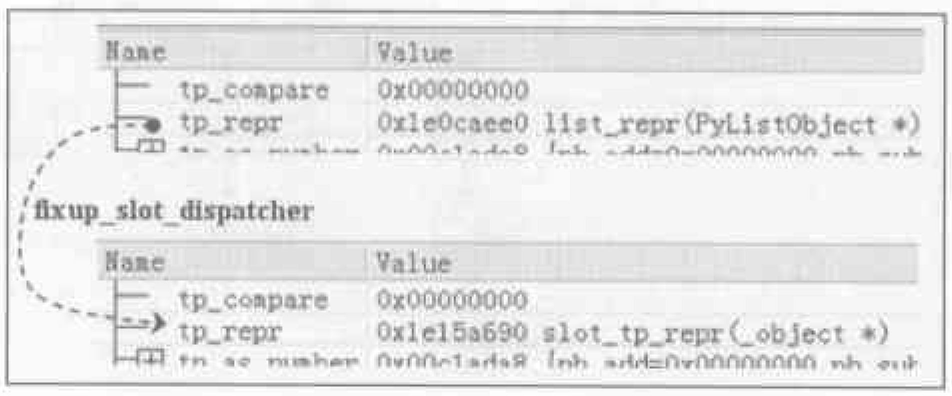



图 12-11 fixup_slot_dispatcher 前后的 tp_repr

#### 12.2.3.4 确定 MRO

所谓的 MRO，即是指 Method Resolve Order，更一般地，也是一个 class 对象的属性

解析顺序。如果 Python 像 Java 一样，只支持单根继承的话，这就不是一个问题了。但是 Python 是支持多重继承的，在多重继承时，就必须设置按照何种顺序解析属性。考虑如下的 Python 代码：

```
[mro.py]
class A(list):
    def show(self):
        print "A::show"

class B(list):
    def show(value):
        print 'B::show'

class C(A):
    pass

class D(C, B):
    pass

d = D()
d.show()
```

由于在 D 的基类 A 和 B 中都实现了 show，那么在调用 d.show() 时，究竟是 A 的 show 被调用，还是 B 的 show 被调用呢？Python 内部在 PyType_Ready 中通过 mro_internal 函数完成了对一个类型的 mro 顺序的建立。Python 虚拟机将创建一个 tuple 对象，在对象中依次存放着一组 class 对象。在 tuple 中，class 对象的顺序就是 Python 虚拟机在解析属性时的 mro 顺序。最终这个 tuple 将被保存在 PyTypeObject.tp_mro 中。

由于 mro_internal 在内部实现时相当繁杂，所以这里我们不深入代码，仅在概念上剖析 Python 虚拟机如何确定一个 class 对象的 mro 列表。

对于在 mro.py 中的 D，Python 虚拟机会在内部创建一个 list，其中根据 D 的声明依次放入 D 和它的基类，如图 12-12 所示：

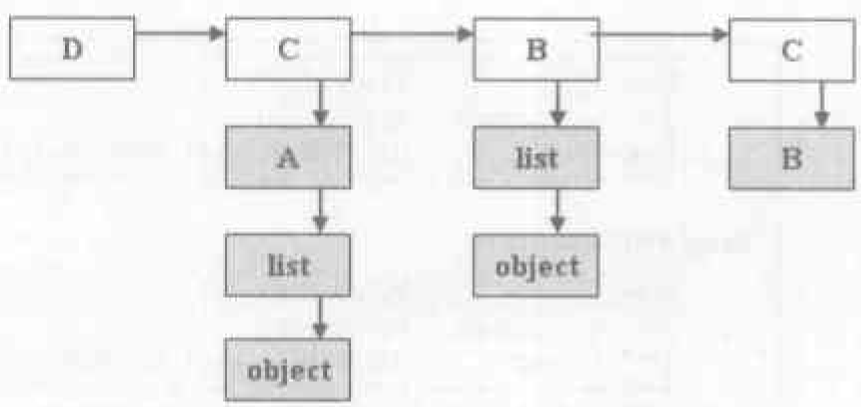



图 12-12 D 建立 mro 列表时 Python 虚拟机内部的辅助 list

注意在 list 的最后一项存放着一个包含所有 D 的直接基类的列表。Python 虚拟机将从左到右遍历该 list，当访问到 list 中的任一个基类时，如果基类存在 mro 列表，则会转而访问基类的 mro 列表。在访问的过程中，不断将所访问到的 class 对象放入到 D 自身

的 mro 列表中去。

我们跟踪这个遍历的过程来看一下。

1. 首先获得 D，D 的 mro 列表（tp_mro）中没有 D，所以放入 D。
2. 获得 C，D 的 mro 列表没有 C，所以放入 C。现在 Python 虚拟机发现 C 中存在 mro 列表，所以转而访问 C 的 mro 列表：
   - 获得 A，D 的 mro 列表中没有 A，放入 A；
   - 获得 list，这里需要特别注意，尽管 D 的 mro 列表中没有 list，但是在后面的 B 的 mro 列表中出现了 list，那么 Python 虚拟机会跳过这里的 list，将 list 的处理推迟到处理 B 的 mro 列表时；
   - 获得 object，同样，将对 object 的处理推迟。
3. 获得 B，D 的 mro 列表中没有 B，所以放入 B。转而访问 B 的 mro 列表：
   - 获得 list，这时可以将 list 放入 D 的 mro 列表；
   - 获得 object，这时可以将 object 放入 D 的 mro 列表。

当遍历的过程结束后，D 的 mro 列表中也就存储了一个 class 对象的顺序列表了。从上面的遍历过程可以看到，这个列表是（D、C、A、B、list、object），我们可以通过在 mro.py 中添加如下的代码，对这个结果进行验证：

```
[mro.py]
……
for t in D.__mro__:
    print t
```

我们还可以改变 D 的基类列表，对比输出结果观察上面提到的确定 mro 列表的算法的正确性。输出结果如表 12-2 所示：

表 12-2 不同继承顺序下的 mro 列表

| class D(B, C) | class D(C, B, A) | class D(C, B) |
|---|---|---|
| <class '__main__.D'><br><class '__main__.B'><br><class '__main__.C'><br><class '__main__.A'><br><type 'list'><br><type 'object'> | <class '__main__.D'><br><class '__main__.C'><br><class '__main__.B'><br><class '__main__.A'><br><type 'list'><br><type 'object'> | <class '__main__.D'><br><class '__main__.C'><br><class '__main__.A'><br><class '__main__.B'><br><type 'list'><br><type 'object'> |

### 12.2.3.5 继承基类操作

Python 虚拟机确定了 mro 列表之后，就会遍历 mro 列表（注意，由于一个 class 对象的 mro 列表的第一项总是其自身，所以遍历是从第二项开始的）。在 mro 列表中实际上

存储的就是 class 对象的所有直接和间接基类，Python 虚拟机会将 class 对象自身没有设置而基类中设置了的操作拷贝到 class 对象中，从而完成对基类操作的继承动作。

这个继承操作的动作发生在 inherit_slots 中：

```
[typeobject.c]
int PyType_Ready(PyTypeObject *type)
{
    ……
    bases = type->tp_mro;
    n = PyTuple_GET_SIZE(bases);
    for (i = 1; i < n; i++) {
        PyObject *b = PyTuple_GET_ITEM(bases, i);
        inherit_slots(type, (PyTypeObject *)b);
    }
    ……
}
```

在 inherit_slots 中，会拷贝相当多的操作，这里我们拿 nb_add 来做个例子：

```
[typeobject.c]
static void inherit_slots(PyTypeObject *type, PyTypeObject *base)
{
    PyTypeObject *basebase;

#define SLOTDEFINED(SLOT) \
     (base->SLOT != 0 && (basebase == NULL || base->SLOT != basebase->SLOT))

#define COPYSLOT(SLOT) \
    if (!type->SLOT && SLOTDEFINED(SLOT)) type->SLOT = base->SLOT

#define COPYNUM(SLOT) COPYSLOT(tp_as_number->SLOT)

    if (type->tp_as_number != NULL && base->tp_as_number != NULL) {
        basebase = base->tp_base;
        if (basebase->tp_as_number == NULL)
            basebase = NULL;
        COPYNUM(nb_add);
        ……
    }
    ……
}
```

我们知道 PyBool_Type 中并没有设置 nb_add 操作，但它的 tp_base 设置的是&PyInt_Type，而 PyInt_Type 中却设置了 nb_add 操作。所以我们可以在 PyType_Ready 中添加输出语句，当处理的 type 分别是 bool 和 int 时，输出其 nb_add 的地址，进行验证。因为按照 inherit_slots 的结果，这两个地址应该都指向同一个地址，即 int_add 的地址。

```
if(strcmp(type->tp_name, "bool") == 0) {
    printf("bool nb_add : 0x%X\n", *(type->tp_as_number->nb_add));
}
if(strcmp(type->tp_name, "int") == 0) {
    printf("int nb_add  : 0x%X\n", *(type->tp_as_number->nb_add));
}
```

输出的结果为：

```
int nb_add  : 0x1E0C3230
bool nb_add : 0x1E0C3230
```

这个结果预示着对于 Python 中的两个 bool 对象，我们可以进行加法操作，读者可以自己尝试一下把两个 bool 对象相加，结果会令你先大吃一惊，但是旋即又会觉得“理当如此”的☺。

#### 12.2.3.6 填充基类中的子类列表

到这里，PyType_Ready 还剩下最后一个重要的动作了：设置基类中的子类列表。在每一个 PyTypeObject 中，有一个 tp_subclasses，这个东西在 PyType_Ready 完成后将是一个 list 对象。在其中存放着所有直接继承自该类型的 class 对象。PyType_Ready 通过调用 add_subclass 完成向这个 tp_subclasses 中填充子类对象的动作：

```
[typeobject.c]
int PyType_Ready(PyTypeObject *type)
{
    PyObject *dict, *bases;
    PyTypeObject *base;
    Py_ssize_t i, n;
    ……

    //填充基类的子类列表
    bases = type->tp_bases;
    n = PyTuple_GET_SIZE(bases);
    for (i = 0; i < n; i++) {
        PyObject *b = PyTuple_GET_ITEM(bases, i);
        add_subclass((PyTypeObject *)b, type);
    }
    ……
}
```

在图 12-13 中我们验证了这个子类列表的存在：

```
>>> int.__subclasses__()
[<type 'bool'>]
>>> object.__subclasses__()
[<type 'type'>, <type 'weakref'>, <type 'int'>, <type 'basestring'>
'>, <type 'NoneType'>, <type 'NotImplementedType'>, <type 'Encoding
'exceptions.BaseException'>, <type 'module'>, <type 'imp.NullImport
zipimport.zipimporter'>, <type 'nt.stat_result'>, <type 'nt.statvfs
ype 'dict'>, <type 'function'>, <type '_types.Helper'>, <class 'sit
 <class 'site._Helper'>, <type 'set'>, <class 'site.Quitter'>, <typ
class 'codecs.IncrementalEncoder'>, <class 'codecs.IncrementalDecod
```

图 12-13 验证 subclasses 的存在

果然，object 真不愧是“万物之母”，那么多的 class 对象都是继承自 object 的。到了这里，我们才算是完整地剖析了 PyType_Ready 的动作，可以看到，Python 虚拟机对 Python 的内置类型对应的 PyTypeObject 进行了多种复杂的改造工作，总结一下，主要包

括以下的工作：

- 设置 type 信息、基类及基类列表；
- 填充 tp_dict；
- 确定 mro 列表；
- 基于 mro 列表从基类继承操作；
- 设置基类的子类列表。

这里列出的 PyType_Type 中的动作序列只是一个框架性的概括，在处理不同类型的时候，有的操作不一定完全等同，比如对于不同类型，可能从基类继承操作时就会有不同的行为，有的类型可能继承得多，有的可能继承得少。对于某个特定类型，读者可以自己跟踪 PyType_Ready 的操作，以清楚了解初始化过程中的每一个细节。

## 12.3 用户自定义 class

从本节开始，我们将正式进入对用户自定义 type（class）的剖析。在 class_0.py 中，我们将研究单个 class 的实现，所以在这里并没有关于继承及多态的讨论。然而在 class_0.py 中，我们看到了许多关于类的内容，其中包括类的定义、类的构造函数、对象的实例化、类成员函数的调用等。这些内容就是在本节中将深入剖析的内容。

```
[class_0.py]
class A(object):
    name = 'Python'
    def __init__(self):
        print 'A::__init__'

    def f(self):
        print 'A::f'

    def g(self, aValue):
        self.value = aValue
        print self.value

a = A()
a.f()
a.g(10)
```

在第 11 章对 Python 中函数机制的分析中，我们知道，对于一个包含有函数定义的 Python 源文件，在 Python 源文件编译之后，会得到一个与源文件对应的 PyCodeObject 对象 A，而与函数对应的 PyCodeObject 对象 B 则存储在 A 的 co_cnosts 变量中。那么对于包含类的 Python 源文件，编译之后的结果又如何呢？

我们可以照葫芦画瓢，根据以前的经验，推测与 class_0.py 对应的 PyCodeObject 对

象会包含一个与 class 对应的 PyCodeObject，而与 class 对应的 PyCodeObject 对象则包含 3 个与函数对应的 PyCodeObject。这正是 Python 编译之后的结果，如图 12-14 所示（图中的方框表示 PyCodeObject 对象，虚线箭头表示包含关系）：

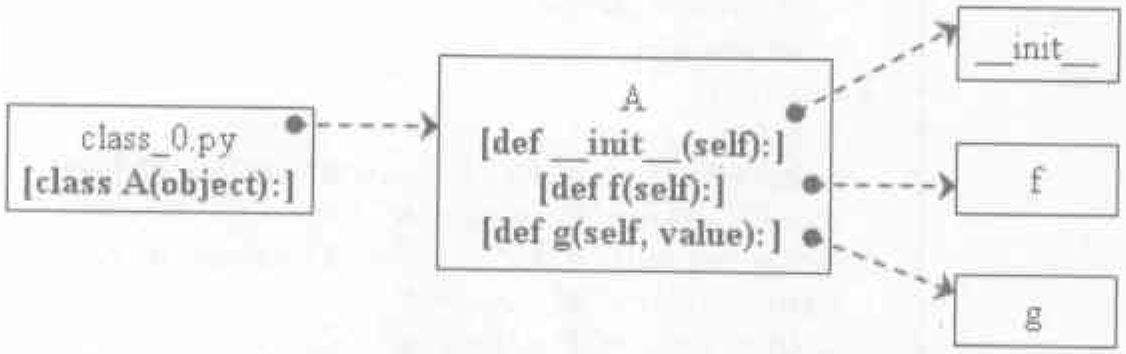



图 12-14 class_0.py 编译后的 PyCodeObject 之间的关系

在第 11 章对函数机制的剖析中，我们已经看到，函数的声明语句 def 语句和函数的实现代码虽然是一个逻辑整体，但是它们对应的字节码指令却是分离在两个 PyCodeObject 中的。在类机制中，同样存在这样的分离现象。声明类的 class A 语句——准确地说，应该是创建类 A 的语句——编译后的字节码指令序列存储在与 class_0.py 对应的 PyCodeObject 对象中；而 class A 的实现代码编译后的字节码指令序列则存储在 A 所对应的 PyCodeObject 对象中。这一点在图 12-14 中可以看得很清楚。在图 12-14 中还可以看到，类的成员函数与一般的函数相同，同样也会有这种声明和实现分离的现象。

当 Python 执行引擎开始执行 class_0.py 时，是从与 class_0.py 对应的那个 PyCodeObject 开始的，第一步就是执行“class A”这条 Python 语句，并创建 class 对象。

## 12.3.1 创建 class 对象

### 12.3.1.1 class 的动态元信息

所谓 class 的元信息就是指关于 class 的信息，比如说 class 的名称，它所拥有的属性、方法，该 class 实例化时要为实例对象申请的内存空间的大小等。对于 class_0.py 中所定义的 class A 来说，我们必须知道这样的信息：在 class A 中，有一个符号 f，这个 f 对应了一个函数；还有一个符号 g，这个 g 也对应了一个函数。有了这些关于 A 的元信息，才能创建 A 的 class 对象，否则，我们是没有办法创建 A 的 class 对象的。元信息在现代编程语言中是一个非常重要的概念，正是有了这个东西，Java、C#中的一些初级的诸如 Reflection 等动态特性才有可能得到实现。在以后的剖析中我们可以看到，在 Python 中，元信息的概念被发挥得淋漓尽致，因而 Python 也提供了 Java、C#等语言所没有的高度灵活的动态性。

在考察具体的字节码之前，我们首先来看看与 class_0.py 对应的 PyCodeObject 中，

常量表（co_consts）和符号表（co_names）的情况，如图 12-15 所示：

```
- <consts>
    <internStr index="0" length="1" value="A" />
  + <codeObject>
    <int value="10" />
    <NoneObject />
  </consts>
- <names>
    <internStr index="11" length="6" value="object" />
    <strRef index="0" value="A" />
    <internStr index="12" length="1" value="a" />
    <strRef index="4" value="f" />
    <strRef index="7" value="g" />
  </names>
```

图 12-15 class_0.py 编译后的符号表和常量表

好了，现在可以深入到字节码中去了，当 Python 开始执行 class_0.py 时，开始的一段字节码如下：

```
[PyCodeObject for class_0.py]
class A(object):
0   LOAD_CONST     0 (A)
3   LOAD_NAME      0 (object)
6   BUILD_TUPLE    1
9   LOAD_CONST     1 (code object for A)
12  MAKE_FUNCTION    0
15  CALL_FUNCTION    0
18  BUILD_CLASS
19  STORE_NAME     1 (A)
```

我们在字节码段的前面标出了这段字节码位于哪个 PyCodeObject 中，这样能够更清晰地展示 Python 虚拟机现在所处的位置。

首先，“0 LOAD_CONST 0”指令将类 A 的名称压入到运行时栈中，而接下来的 LOAD_NAME 指令和 BUILD_TUPLE 指令是一个非常关键的点，这两条指令将基于类 A 的所有基类创建一个基类列表，当然这里只有一个名为 object 的基类。

随后，Python 虚拟机通过“9 LOAD_CONST 1”指令将与 A 对应的 PyCodeObject 压入到运行时栈中，并通过 MAKE_FUNCTION 指令创建一个 PyFunctionObject 对象。在这些操作完成之后，我们来看一看这时的运行时栈，如图 12-16 所示：

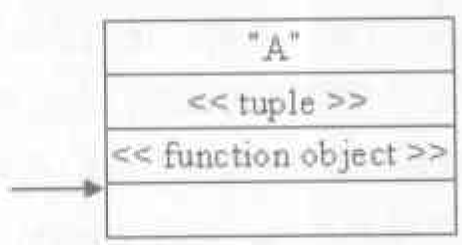



图 12-16 MAKE_FUNCTION 指令完成后的运行时栈

随后，Python 虚拟机开始执行“15 CALL_FUNCTION 0”指令。从前面对函数机制的分析我们已经看到，对 CALL_FUNCTION 的执行最终将创建一个新的 PyFrameObject 对象，并开始执行这个 PyFrameObject 对象中所包含的字节码序列，很显然，这些字节码序列来自在运行时栈中静静伫立的那个 PyFunctionObject 对象。参考上面的描述，我们可以发现，这段字节码序列实际上就是来自与 A 对应的 PyCodeObject 对象的。换句话说，现在 Python 虚拟机所面对的目标从与 class_0.py 对应的字节码序列转换到了与 class A 对应的字节码序列：

```
[PyCodeObject for class A]
0   LOAD_NAME  0 (__name__)
3   STORE_NAME   1 (__module__)
name = 'Python'
6   LOAD_CONST   0 ('Python')
9   STORE_NAME   2 (name)
def __init__(self):
12   LOAD_CONST    1 (code object for function __init__)
15   MAKE_FUNCTION    0
18   STORE_NAME    3 (__init__)
def f(self):
21   LOAD_CONST    2 (code object for function f)
24   MAKE_FUNCTION    0
27   STORE_NAME    4 (f)
def g(self, aValue):
30   LOAD_CONST    3 (code object for function g)
33   MAKE_FUNCTION    0
36   STORE_NAME    5 (g)

39   LOAD_LOCALS
40   RETURN_VALUE
```

Python 执行引擎在执行 CALL_FUNCTION 时，实际上只是执行了 1 个赋值语句和 3 个 def 语句，创建了 3 个 PyFunctionObject 对象，创建这 3 个对象有什么用呢？别急，我们先看一看，与 class A 对应的 PyCodeObject 对象中的常量表和符号表，如图 12-17 所示：

```
- <consts>
    <internStr index="1" length="6" value="Python" />
  + <codeObject>
  + <codeObject>
  + <codeObject>
  </consts>
- <names>
    <internStr index="8" length="8" value="__name__" />
    <internStr index="9" length="10" value="__module__" />
    <internStr index="10" length="4" value="name" />
    <strRef index="3" value="__init__" />
    <strRef index="4" value="f" />
    <strRef index="7" value="g" />
  </names>
```

图 12-17 A 编译后的符号表和常量表

开始的 LOAD_NAME 和 STORE_NAME 将符号“__module__”和全局名字空间中符号“__name__”对应的值（“__main__”）关联了起来，并放入到 local 名字空间（PyFrameObject 对象的 f_locals）中。需要说明的是，与之前看到的函数机制不同，在前面执行“15 CALL_FUNCTION 0”指令，创建新的 PyFrameObject 对象时，PyFrameObject 中的 f_locals 被创建了，并指向了一个 PyDictObject 对象。而在函数机制中，这个 f_locals 是被设置为 NULL 的，函数机制中局部变量是以一种位置参数的形式放在了运行时栈前面的那段内存中。如果记不太清了，请参考第 11 章对函数机制的剖析。

接着，Python 虚拟机连续执行了 3 个（LOAD_CONST、MAKE_FUNCTION、STORE_NAME）指令序列对，每个指令序列都会创建一个与类中成员函数对应的 PyFunctionObject 对象，并将函数名和其对应的 PyFunctionObject 对象通过 STORE_NAME 存入到 local 名字空间中。

到了这时，回头看一看，这一路下来，我们好像创建了很多东西。冷静一下，想一想，到目前为止创建的有用的东西都被放到了 local 名字空间中，这里面存放的恰恰是最重要的东西——class A 的元信息。

更准确地说，现在 local 名字空间中存放的应该是 class A 的动态元信息，既然有了动态元信息，那么必然会有静态元信息。关于动态元信息和静态元信息的区别，我们将在后面给出。想象一下，当你在 class A 中寻找 f 时，这个 f 对应的究竟是阿猫，还是阿狗呢？只有有了这个 local 名字空间，才能知道，这个 f 对应的竟然是一个函数。

好了，现在我们手心里捏着 class A 的动态元信息，但不对呀，我们要的应该是一个与 A 对应的 class 对象呀。难道说现在所使用到的 PyFrameObject 对象就是代表 class A 的 class 对象吗？看上去好像不太对劲啊，别着急，中国有句老话，“退一步海阔天空”。现在，我们开始向后退，一直要退到我们出发的地方，还记得吗？就是那个 class_0.py 的 PyCodeObject 对象中的“15 CALL_FUNCTION 0”指令。但是在后退之前，我们要顺手牵羊，将好不容易得来的 class A 的动态元信息，也就是当前 PyFrameObject 中的 f_locals 带走。徐志摩是不带走一片云彩的，但是我们必须把 local 名字空间带走，否则，以后只能喝西北风了。

```
[LOAD_LOCALS]
        if ((x = f->f_locals) != NULL) {
            PUSH(x);
            continue;
        }
```

LOAD_LOCALS 将 f_locals 压入运行时栈中，而随后的 RETURN_VALUE 指令将处于栈顶的 f_locals 一脚踢出，踢给之前那个古老的 CALL_FUNCTION：

```
[CALL_FUNCTION]
        PyObject **sp;
```

```
            sp = stack_pointer;
            x = call_function(&sp, oparg);
            stack_pointer = sp;
            PUSH(x);
```

CALL_FUNCTION 将接过这个惊险的长传，然后将 f_locals 压入到运行时栈中，现在的运行时栈如图 12-18 所示：

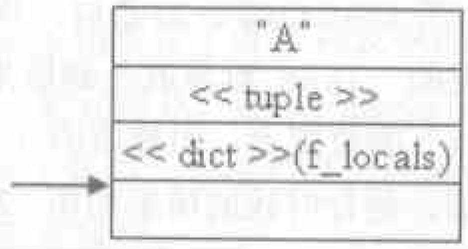



图 12-18 CALL_FUNCTION 指令完成后的运行时栈

我们可以在 CALL_FUNCTION 的实现代码中添加打印 PyDictObject 对象的代码，以观察返回的对象，如图 12-19 所示：

```
>>> class A(object):
        a = 1
        d = {1:'Robert', 2:'Python'}
        def f(self):
            pass
        def g(self, value):
            pass

============== Object x in CALL_FUNCTION ==============
a : 1
__module__ : __main__
d : {1: 'Robert', 2: 'Python'}
g : <function g at 0x00DC8430>
f : <function f at 0x00DB33B0>
```

图 12-19 探测 CALL_FUNCION 指令代码中的 x

### 12.3.1.2 metaclass

在成功地获得了 class A 的动态元信息之后，CALL_FUNCTION 的使命也结束了，随后,指令“18 BUILD_CLASS”开始被激活，Python 执行引擎从现在开始正式进入创建 class 对象的动作：

```
[BUILD_CLASS]
        u = TOP();     //class 的动态元信息 f_locals
        v = SECOND();  //class 的基类列表
        w = THIRD();   //class 的名 "A"
        STACKADJ(-2);
        x = build_class(u, v, w);
        SET_TOP(x);
        Py_DECREF(u);
```

```
        Py_DECREF(v);
        Py_DECREF(w);
```

对照图 12-18，u、v、w 分别是什么对象，现在已经很清晰了。在获得了 class 动态元信息（也就是 class 的属性表啦）、class 名称和 class 基类列表之后，Python 虚拟机会调用 build_class 创建 class 对象，然后将创建的 class 对象压入到运行时栈中。

需要特别注意的是，在 STACKADJ(-2)之后，栈顶指针已经向上移动了两个位置，这样，SET_TOP 就会将创建的 class 对象放入到原来运行时栈中存储表示 class 名称的 PyStringObject 对象“A”的位置上，而最后的 3 个 Py_DECREF 保证了这种操作的安全性。在 BUILD_CLASS 之后，运行时栈的情形如图 12-20 所示：

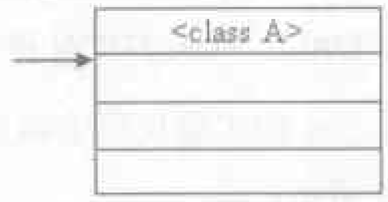



图 12-20 BUILD_CLASS 指令结束后的运行时栈

在 BUILD_CLASS 指令结束后，会通过“19 STORE_NAME 1”指令将（“A”，<class A>）存放到 local 名字空间中。到此，符号 A 与其对应的 class 对象之间的联系已经完全建立起来了，Python 虚拟机在以后的执行中可以正常地使用 A 这个符号了。这里我们所谓的“使用符号 A”，嗯，没错，就是将<class A>这个 class 对象实例化，创建 instance 对象。不过对创建 instance 对象的剖析将留到下一节中。现在我们还是回过头来看看那个神秘的 build_class。在 build_class 中究竟发生了什么呢？Python 又是用怎样一个对象来表示 class 的呢？别急，我们马上就进入 build_class 中。

### 12.3.1.2.1 获得 metaclass

**代码清单 12-3**

```
[ceval.c]
PyObject * build_class(PyObject *methods, PyObject *bases, PyObject *name)
{
    PyObject *metaclass = NULL, *result, *base;

    //[1] : 检查属性表中是否有指定的__metaclass__
    if (PyDict_Check(methods))
        metaclass = PyDict_GetItemString(methods, "__metaclass__");
    if (metaclass != NULL)
        Py_INCREF(metaclass);
    else if (PyTuple_Check(bases) && PyTuple_GET_SIZE(bases) > 0) {
        //[2] : 获得 A 的第一基类，object
        base = PyTuple_GET_ITEM(bases, 0);
        //[3] : 获得 object.__class__
        metaclass = PyObject_GetAttrString(base, "__class__");
    }
    else {
```

```
        ......
    }
    result = PyObject_CallFunctionObjArgs(metaclass, name, bases, methods, NULL);
    ......
    return result;
}
```

在前面，Python 虚拟机获得了关于 class 的属性表（动态元信息），在 build_class 中，这个动态元信息作为 methods 出现在了参数列表中。有一点值得注意的是，methods 中并没有包含所有的关于 class 的元信息，在 methods 中，只包含了在 class 中包含什么属性，什么方法。由于从广义上来讲，方法也是一种属性，所以我们可以说，class 的动态元信息中包含了 class 的所有属性。

但是，对于这个 class 对象的 type 是什么，应该如何创建，要分配多少内存，却没有任何的信息。在 build_class 中，metaclass 正是关于 class 对象的另一部分元信息，我们称之为静态元信息，也就是在图 12-3 中需要放到最左侧的 metaclass 对象。在静态元信息中，隐藏着所有的 class 对象应该如何创建的信息，注意，我们这里说地是**所有的** class 对象。

在 build_class 中，实际上包含了为 classic class 和 new style class 确定 metaclass 的过程，当然这里我们只考虑为 new style class 确定 metaclass 的过程。

首先，Python 虚拟机在代码清单 12-3 的[1]处会检查用户在定义 class 时是否指定了 __metaclass__，如果指定了，当然就使用用户自己指定的 metaclass。

如果用户没有指定，那么 Python 虚拟机会选择 class 的第一基类的 type 作为该 class 的 metaclass。

对于这里的 A 来说，其第一基类（事实上，A 只有一个基类）为 object，而我们已经熟悉 object.__class__ = <type 'type'>。所以最终获得的 metaclass 为<type 'type'>这个 class 对象。

对于 PyIntObject、PyDictObject 这些对象，其所有的元信息都包含在其对应的类型对象中。而为什么关于一个 class 对象的所有元信息不能包含在 metaclass 中，却要分离为两部分呢？因为，用户会在.py 源文件中定义不同的 class，其所包含的属性肯定是不相同的，这就决定了只能使用动态的机制来保存 class 的属性，这个元信息只能是动态的，所以我们称为动态元信息，即我们看到的参数 methods；而对于所有的 class 都可能共用的元信息，比如 class 对象的 type 和 class 对象的创建策略，这些则存放在了 class 对象的 metaclass 中。

PyIntObject、PyDictObject 这些对象是 Python 静态提供的，它们都具有相同的接口集合，当然，有的对象可能是不支持某个接口的，但这不影响它们的所有元信息可以完

全存储在其类型对象中；而用户自定义的 class 对象，其接口集合是动态的，不可能在 metaclass 中静态地指定。图 12-21 展示了多个 class 对象和元信息的关系。

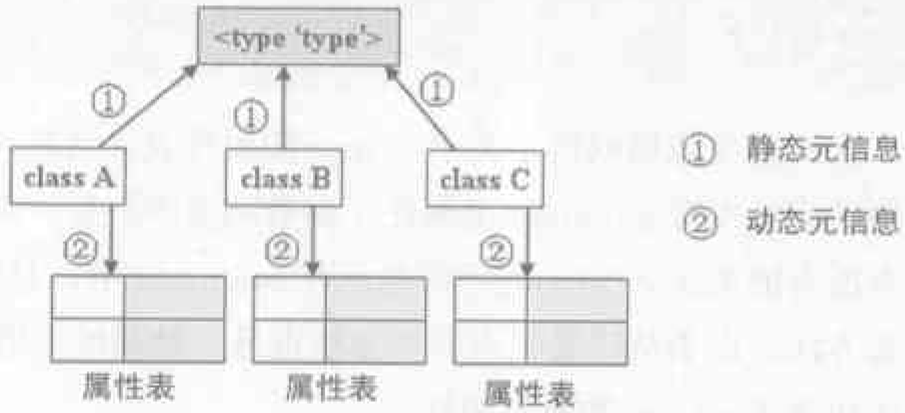



图 12-21 class 对象与元信息之间的关系

#### 12.3.1.2.2 调用 metaclass

获得了 metaclass 之后，build_class 通过 PyObject_CallFunctionObjArgs 函数完成"调用 metaclass"的动作，从而完成 class 对象的创建。

传递到 PyObject_CallFunctionObjArgs 中的对象是<type 'type'>（也就是 PyType_Type），而 PyType_Type 作为 metaclass 的功能正是通过它的可调用性体现出来的。换句话说，我们调用一个函数，得到一些输出，同样，我们调用 PyType_Type，得到一个 class 对象。

现在问题来了，一个 Python 程序中 class 可能成千上万，而 PyType_Type 却只有一个，这一个 PyType_Type 如何创建出不同的 class 对象呢？其中的奥妙则集中在 PyObject_CallFunctionObjArgs 的其他几个参数中。我们已经知道，这几个参数分别是 class 的类名、基类列表和属性列表，在 PyObject_CallFunctionObjArgs 中，这几个参数会被打包到一个 tuple 对象中，最终进入 PyObject_Call 函数。好了，现在我们开始进入创建 class 对象的幽深处：

```
[object.h]
typedef PyObject * (*ternaryfunc)(PyObject *, PyObject *, PyObject *);

[abstract.c]
PyObject* PyObject_Call(PyObject *func, PyObject *arg, PyObject *kw)
{
    //arg 即是 PyObject_CallFunctionObjArgs 中打包得到的 tuple 对象
    ternaryfunc call = func->ob_type->tp_call;
    PyObject *result = (*call)(func, arg, kw);
    return result;
}
```

最终，由于 PyType_Type 的 ob_type 还是指向 PyType_Type，所以最终将调用到 PyType_Type 中定义的 tp_call 操作。下面来看一看 PyType_Type 的 tp_call 操作：

```
[typeobject.c]
type_call(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    PyObject *obj;

    obj = type->tp_new(type, args, kwds);
    ……//如果创建的是实例对象，则调用"__init__"进行初始化
    return obj;
}
```

PyType_Type 中的 tp_new 指向 type_new，而这个 type_new 才是 class 对象创建的第一案发现场。由于 type_new 的代码相当繁杂，所以这里我们做了相当的简化（代码清单 14-2），有兴趣的读者请参考 Python 源码。

**代码清单 12-4**

```
[typeobject.c]
static PyObject * type_new(PyTypeObject *metatype, PyObject *args, PyObject
*kwds)
{
    //metatype 是 PyType_Type(<type 'type'>), args 中包含了(类名, 基类列表, 属性表)
     PyObject *name, *bases, *dict;
     static char *kwlist[] = {"name", "bases", "dict", 0};
     PyTypeObject *type, *base, *tmptype, *winner;
     PyHeapTypeObject *et;
     Py_ssize_t slotoffset;

    //将 args 中的(类名, 基类列表, 属性表)分别解析到 name, bases, dict 三个变量中
    PyArg_ParseTupleAndKeywords(args, kwds, "SO!O!:type", kwlist,
                    &name,
                    &PyTuple_Type, &bases,
                    &PyDict_Type, &dict))

    ......//确定最佳 metaclass, 存储在 PyObect *metatype 中
    ......//确定最佳 base, 存储在 PyObject *base 中

    //为 class 对象申请内存
    //尽管 PyType_Type 为 0, 但 PyBaseObject_Type 的为 PyType_GenericAlloc,
    //在 PyType_Ready 中被继承了
    //创建的内存大小为 tp_basicsize + tp_itemsize
     type = (PyTypeObject *)metatype->tp_alloc(metatype, nslots);
     et = (PyHeapTypeObject *)type;
     et->ht_name = name;

    //设置 PyTypeObject 中的各个域
     type->tp_as_number = &et->as_number;
     type->tp_as_sequence = &et->as_sequence;
     type->tp_as_mapping = &et->as_mapping;
     type->tp_as_buffer = &et->as_buffer;
     type->tp_name = PyString_AS_STRING(name);

    //设置基类和基类列表
    type->tp_bases = bases;
```

```
    type->tp_base = base;

    //设置属性表
    type->tp_dict = dict = PyDict_Copy(dict);

    //如果自定义 class 中重写了__new__，将__new__对应的函数改造为 static 函数
    tmp = PyDict_GetItemString(dict, "__new__");
    tmp = PyStaticMethod_New(tmp);
    PyDict_SetItemString(dict, "__new__", tmp);

    //[1]: 为 class 对象对应的 instance 对象设置内存大小信息
    slotoffset = base->basicsize
    type->tp_dictoffset = slotoffset;
    slotoffset += sizeof(PyObject *);
    type->tp_weaklistoffset = slotoffset;
    slotoffset += sizeof(PyObject *);
    type->tp_basicsize = slotoffset;
    type->tp_itemsize = base->tp_itemsize;
    ......

    //调用 PyType_Ready(type)对 class 对象进行初始化
    PyType_Ready(type);
    return (PyObject *)type;
}
```

Python 虚拟机首先会将类名、基类列表和属性表从 tuple 对象中解析出来，然后会基于基类列表及传入的 metaclass（参数 metatype）确定最佳的 metaclass 和 base，对于我们的 A 来说，最佳 metaclass 为 <type 'type'>，最佳的 base 为 <type 'object'>。

随后，Python 虚拟机会调用 metatype->tp_alloc 尝试为所要创建的与 A 对应的 class 对象分配内存。这里需要注意的是，在 PyType_Type 中，我们会发现 tp_alloc 为 NULL，这很奇怪，对吧？但是别忘了，之前我们已经提到，在 Python 进行初始化时，会对所有的内置 class 对象通过 PyType_Ready 进行初始化，在这个初始化过程中，有一项动作就是从基类继承各种操作。由于 type.__bases__中的第一基类是 <type 'object'>，所以 <type 'type'>会继承 <type 'object'>的 tp_alloc 操作，即 PyType_GenericAlloc。对于我们的 A（或者说，对于任何继承自 object 的 class 对象来说）PyType_GenericAlloc 最终将申请 metatype->tp_basicsize + metatype->tp_itemsize 大小的内存空间。从 PyType_Type 的定义中我们可以看到，这个大小实际上就是 sizeof (PyHeapTypeObject) +sizeof (PyMemberDef)。到这里,有种天开云散见晴空的感觉了吧，当初就觉得 PyHeapTypeObject 这个东西怎么莫名其妙地在 Python 中就出现了呢，原来 PyHeapTypeObject 是为用户自定义的 class 对象准备的。

此后，就是设置<class A>这个 class 对象的各个域，其中包括了在 tp_dict 上设置了属性表。需要特别关注代码清单 12-4 的[1]处，这里计算了与 <class A> 对应的 instance 对象的内存大小信息，换句话说，当我们以后通过 a = A()这样的表达式创建

一个 instance 对象时，需要为这个 instance 对象申请多大的内存呢？对于 A（对任何继承自 object 的 class 对象也成立）来说，这个大小为 PyBaseObject_Type->tp_basicsize + 8。其中的 8 为 2*sizeof(PyObject*)。为什么后面要跟着两个 PyObject* 的空间，而且这些空间的地址被设置给了 tp_dictoffset 和 tp_weaklistoffset 呢？这里先不表，留待以后详解。

最后，Python 虚拟机还会调用 PyType_Ready 对 <class A> 进行和内置 class 对象一样的初始化动作。到此，A 对应的 class 对象正式创建完毕。图 12-22 显示了用户自定义的 class 对象和内置的 class 对象最终在内存布局上的区别。

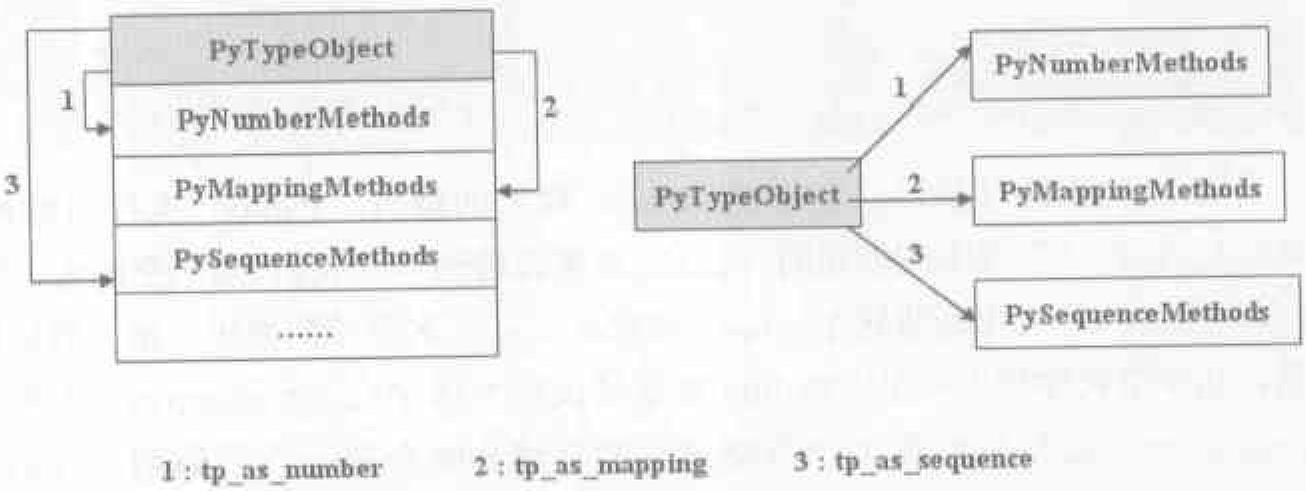



图 12-22 用户自定义 class 对象和内置 class 对象的内存布局对比

本质上，无论是用户自定义的 class 对象还是内置的 class 对象，在 Python 虚拟机内部，都可以用一个 PyTypeObject 来表示。但不同的是，内置 class 对象的 PyTypeObject 及与其关联的 PyNumberMethods 等的内存位置都是在编译时确定的，它们在内存中的位置是分离的；而用户自定义的 class 对象的 PyTypeObject 和 PyNumberMethods 等的内存位置是连续的，必须在运行时动态申请内存。

现在我们对 Python 中“可调用（callable）”这个概念应该是有一定的感性认识了。在 Python 中，这个概念实际上是一个相当通用的概念，不拘对象，不拘大小，只要对象定义了 tp_call 操作，就能进行“调用”操作。我们已经看到，Python 中的 class 对象是“调用”metaclass 对象（比如 <type 'type'>）创建的。如果按照这个逻辑小小地向前推测一步，那么调用 class 对象，是不是就能得到 instance 对象呢？欲知后事如何，且听下节分解☺。

## 12.4 从 class 对象到 instance 对象

嘿，朋友，现在是不是已经头脑发张，两眼发晕了？前面罗罗嗦嗦说了几十页，这个 class 机制到底何时到头啊？这里，我有一个坏消息，一个好消息。

坏消息是，很不幸，我们费尽千辛万苦，创建了 class 对象，仅仅是万里长征走完了第一步。当 Python 虚拟机执行时，在内存中叱咤风云的是一个个 instance 对象，而 class 对象只是幕后英雄。

好消息是我们到了一个关键的转折点，在本节中，我们来看看从 class 对象出发，创建 instance 对象的工作是如何完成的。同时，从这里往后，路会一马平川，不像之前的内容那样摧残大脑☺。

在 class_0.py 中，创建 instance 的动作如下：

```
[PyCodeObject for class_0.py]
a = A()
22   LOAD_NAME      1 (A)
25   CALL_FUNCTION   0
28   STORE_NAME    2 (a)
```

在上一节我们看到，在创建 class 对象的最后，Python 执行引擎通过指令“19 STORE_NAME 1”将刚刚创建的 class 对象存放到了 local 名字空间中，所以指令“22 LOAD_NAME 1”会重新将这个 class 对象从 local 名字空间取出，压入到运行时栈中。显然，从字节码序列中可见，Python 将通过执行“25 CALL_FUNCTION 0”指令来创建一个 instance，这与我们在上一节结束时的猜测是吻合的，即“**调用 class 对象将创建 instance 对象**”。Python 执行引擎在创建了 instance 对象之后，会通过指令“28 STORE_NAME 2”将(a, instance)存放到 local 名字空间中，所以，在这段字节码指令序列完成之后，local 名字空间如图 12-23 所示：

| | |
|---|---|
| "A" | <<class>> |
| "a" | <<instance>> |

图 12-23 创建 instance 对象后的 local 名字空间

在 CALL_FUNCTION 中，Python 同样会沿着 call_function->do_call->PyObject_Call 的调用路径进入到 PyObject_Call 中。前面说过，所谓“调用”，就是执行对象的 type 所对应的 class 对象的 tp_call 操作。所以，在 PyObject_Call，Python 执行引擎会寻找 class 对象<class A>的 type 中定义的 tp_call 操作。<class A>的 type 为<type 'type'>，所以最终将调用 tp_call，在 PyType_Type.tp_call 中又调用了 A.tp_new 是用来创建 instance 对象的。

这里需要特别注意，在创建<class A>这个 class 对象时，Python 虚拟机调用 PyType_Ready 对<class A>进行了初始化，其中的一项动作就是继承基类的操作，所以 A.tp_new 实际上也就是 object.tp_new，在 PyBaseObject_Type 中，这个操作被定义为 object_new。创建 class 对象和创建 instance 对象的不同之处正是在于 tp_new 不同。创建 class 对象，Python 虚拟机使用的是 type_new；而对于 instance 对象，Python 虚拟机则使用 object_new。

在 object_new 中，调用了 A.tp_alloc，这个操作也是从 object 继承而来的，是 PyType_GenericAlloc。前面我们已经提到，PyType_GenericAlloc 最终将申请 A.tp_basicsize + A.tp_itemsize 大小的内存空间。上一节中，这两个量的计算结果为 A.tp_basicsize = PyBaseObject_Type.tp_basicsize + 8 = sizeof(PyObject) + 8 = 24；A.tp_itemsize = PyBaseObject_Type.tp_itemsize = 0。原来，object_new 的所有工作就是申请了一个 24 字节的内存空间。

在申请了 24 字节的内存空间，回到 type_call 之后，由于创建的不是 class 对象，而是 instance 对象，type_call 会尝试进行初始化的动作。

```
[typeobject.c]
static PyObject * type_call(PyTypeObject *type, PyObject *args, PyObject
 *kwds)
{
    PyObject *obj;
    obj = type->tp_new(type, args, kwds);
    type = obj->ob_type;
    type->tp_init(obj, args, kwds);
    return obj;
}
```

对于基于<class A>创建的 instance 对象 obj，其 ob_type 当然也在 PyType_GenericAlloc 中被设置为指向<class A>，其 tp_init 在 PyType_Ready 时会继承 PyBaseObject_Type 的 object_init 操作，但正如在本章 2.3.3 节中描述的那样，因为 A 的定义中重写了__init__，所以在 fixup_slot_dispatchers 中，tp_init 会指向 slotdefs 中指定的与"__init__"对应的 slot_tp_init：

```
[typeobject.c]
static int slot_tp_init(PyObject *self, PyObject *args, PyObject *kwds)
{
    static PyObject *init_str;
    PyObject *meth = lookup_method(self, "__init__", &init_str);

    PyObject_Call(meth, args, kwds);
    return 0;
}
```

在执行 slot_tp_init 时，Python 虚拟机会首先通过 lookup_method 在 class 对象及其 mro 列表中搜索属性"__init__"对应的操作，然后通过 PyObject_Call 调用该操作。属性搜索是一个以后会剖析的独立主题，这里先按下不表。如果你在定义 class 时，重写了__init__操作，那么搜索的结果就是你写的操作，如果没有重写，那么最终的结果将是调用 object_init，在 object_init 中，Python 虚拟机什么也不做，直接返回，所以，当我们通过 a = A()创建一个 instance 对象时，实际上是没有进行任何初始化的动作。

到了这里，我们可以小结一下，从 class 对象创建 instance 对象的两个步骤：

- instance = class.__new__(class, args, kwds)
- class.__init__(instance, args, kwds)

其中，args 为一个 tuple 对象，里面包含着创建 instance 对象的各种参数，而 kwds 通常为 NULL。需要特别注意的是，这两个步骤也适用于从 metaclass 对象创建 class 对象。从 metaclass 对象创建 class 对象的过程也是一个从 class 对象创建 instance 对象的过程，前面我们就已经提过，class 对象具有二相性。

## 12.5 访问 instance 对象中的属性

在前面的章节我们讨论名字空间时就提到，在 Python 中，形如 x.y 或 x.y()形式的表达式称为“属性引用”，其中 x 为对象，而 y 为对象的属性，这个属性，有可能只是简单的数据，比如字符串或整数，当然，也有可能是成员函数这类比较复杂的东西。

在 class_0.py 中，我们一共调用了两个 class A 的成员函数，一个是不需要参数的成员函数，而另一个是需要参数的成员函数。这里，我们先来看看，对于不需要参数的成员函数，其调用过程是怎样的：

```
[PyCodeObject for class_0.py]
a.f()
31   LOAD_NAME     2 (a)
34   LOAD_ATTR     3 (f)
37   CALL_FUNCTION   0
40   POP_TOP
```

Python 虚拟机通过指令“31 LOAD_NAME 1”，会将 local 名字空间中符号“a”对应的 instance 对象压入到运行时栈中。随后的指令“34 LOAD_ATTR 2”是属性访问机制的关键所在，它会从<instance a>中获得与符号“f”对应的对象，还能想起是个什么对象吗？没错，是一个 PyFunctionObject 对象：

```
[LOAD_ATTR]
          w = GETITEM(names, oparg);
          v = TOP();
          x = PyObject_GetAttr(v, w);
          Py_DECREF(v);
          SET_TOP(x);
```

其中 w 为 PyStringObject 对象“f”，而 v 为运行时栈中的那个 instance 对象<instance a>，从<instance a>中获得“f”对应对象的关键就在 PyObject_GetAttr 身上（见代码清单 12-5）。

代码清单 12-5

```
[object.c]
PyObject* PyObject_GetAttr(PyObject *v, PyObject *name)
{
```

```
    PyTypeObject *tp = v->ob_type;
    //[1]: 通过tp_getattro获得属性对应对象
    if (tp->tp_getattro != NULL)
        return (*tp->tp_getattro)(v, name);

    //[2]: 通过tp_getattr获得属性对应对象
    if (tp->tp_getattr != NULL)
        return (*tp->tp_getattr)(v, PyString_AS_STRING(name));
    //[3]: 属性不存在，抛出异常
    PyErr_Format(PyExc_AttributeError,"'%.50s' object has no attribute
            '%.400s'",tp->tp_name, PyString_AS_STRING(name));
    return NULL;
}
```

在 Python 的 class 对象中，定义了两个与访问属性相关的操作：tp_getattro 和 tp_getattr。其中的 tp_getattro 是首选的属性访问操作，而 tp_getattr 在 Python 中已不再推荐使用。它们之间的区别其实在 PyObject_GetAttr 中已显示得非常清楚，主要是在属性名的使用上，tp_getattro 所使用的属性名必须是一个 PyStringObject 对象，而 tp_getattr 使用的属性名必须是一个 C 中的原生字符串。如果某个类型同时定义了 tp_getattr 和 tp_getattro 两种属性访问操作，那么 PyObject_GetAttr 将优先使用 tp_getattro 操作。

在 Python 虚拟机创建<class A>时，会从 PyBaseObject_Type 中继承其 tp_getattro——PyObject_GenericGetAttr。所以 Python 虚拟机在这里会进入 PyObject_GenericGetAttr。

在 PyObject_GenericGetAttr 中，有一套复杂地确定访问的属性的算法，下面我们以 a.f 为例，用 Python 伪代码来描述这个属性访问算法（由于这里涉及到了 descriptor，第一遍看不明白时，没关系，继续向后看，到了解 descriptor 之后，再回到这里，一切就云开雾散了）：

```
#首先寻找'f'对应的descriptor (descriptor在之后会细致剖析)
#注意: hasattr会在<class A>的mro列表中寻找符号'f'
if hasattr(A, 'f'):
    descriptor = A.f

type = descriptor.__class__
if hasattr(type, '__get__') and (hasattr(type, '__set__') or 'f' not in
  a.__dict__):
    return type.__get__(descriptor, a, A)

#通过descriptor访问失败，在instance对象自身__dict__中寻找属性
if 'f' in a.__dict__:
    return a.__dict__['f']

#instance对象的__dict__中找不到属性，返回a的基类列表中某个基类里定义的函数
#注意: 这里的descriptor实际上指向了一个普通函数
if descriptor:
    return descriptor.__get__(descriptor, a, A)
```

我们通过一段代码来验证这个伪代码描述的算法：

```
class A(object):
  def func(self):
      pass

a = A()
a.func = 1
print a.func
```

很不幸的是，你会发现输出的结果为 1，看上去上面伪代码描述的算法不对呀。实际上，在那段伪代码中，有一个关键的概念——descriptor。在一个 class 中，并不是随意定义一个函数就是 descriptor 了，所以导致输出结果为 1。那么究竟什么才是 descriptor 呢，稍等片刻，谜底一会就能揭开了。

## 12.5.1 instance 对象中的__dict__

在属性访问算法中，我们看到有"a.__dict__"这样的形式。这一点相当奇怪了，在第 12.4 节的描述中，我们看到，从<class A>创建<instance a>时，Python 虚拟机仅仅是为 a 申请了 16 字节的内存，并没有额外的创建 PyDictObject 对象的动作呀。不过在<instance a>中，24 个字节的前 8 个字节是 PyObject，后 8 个字节是为两个 PyObject* 申请的，难道谜底就在这多出的这两个 PyObject*中？

在创建<class A>时，我们曾说到，Python 虚拟机设置了一个名为 tp_dictoffset 的域，从名字上推断，这个可能就是 instance 对象中__dict__的偏移位置。在图 12-24 中，我们展示了我们的猜想。

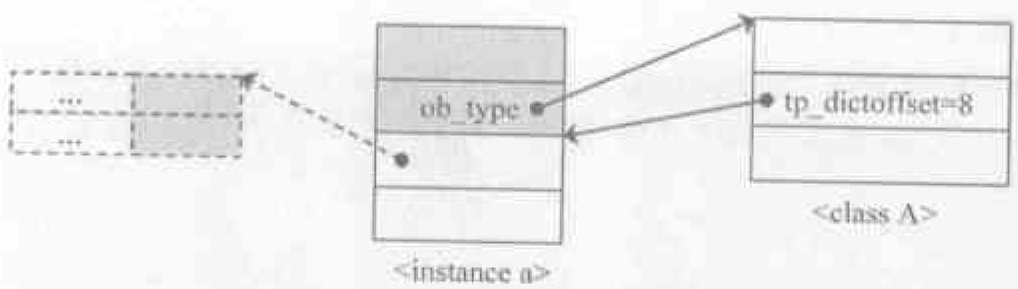



图 12-24 猜想中的 a.__dict__

图 12-24 中，虚线画出的 dict 对象就是我们期望中的 a.__dict__。这个猜想可以在 PyObject_GenericGetAttr 中与伪代码中对应的 C 代码得到证实：

```
[object.c]
PyObject * PyObject_GenericGetAttr(PyObject *obj, PyObject *name)
{
    PyTypeObject *tp = obj->ob_type;
    PyObject *res = NULL;
    Py_ssize_t dictoffset;
    PyObject **dictptr;
```

```
    //inline _PyObject_GetDictPtr 函数的代码
    dictoffset = tp->tp_dictoffset;
    if (dictoffset != 0) {
        PyObject *dict;
        if (dictoffset < 0) {
            …… //处理变长对象
        }
        dictptr = (PyObject **) ((char *)obj + dictoffset);
        dict = *dictptr;
        res = PyDict_GetItem(dict, name);
    }
    ……
}
```

如果 dictoffset 小于 0，意味着 A 是继承自 str 这样的变长对象，Python 虚拟机会对 dictoffset 进行一些处理，最终仍然会使 dictoffset 指向 a 的内存中额外申请的位置。而 PyObject_GenericGetAttr 正是根据这个 dictoffset 获得了一个 dict 对象。

更进一步，查看函数 g 中有设置 self.value 的代码，这个 instance 对象的属性设置动作也会访问 a.__dict__，而且，这个设置动作最终调用的 PyObject_GenericSetAttr 也是 a.__dict__最初被创建的地方：

```
[object.c]
int PyObject_GenericSetAttr(PyObject *obj, PyObject *name, PyObject *value)
{
    PyTypeObject *tp = obj->ob_type;
    PyObject **dictptr;
    ……
    dictptr = _PyObject_GetDictPtr(obj);
    if (dictptr != NULL) {
        PyObject *dict = *dictptr;
        if (dict == NULL && value != NULL) {
            //这里创建了 instance 对象中的__dict__
            dict = PyDict_New();
            *dictptr = dict;
        }
        ……
    }
    ……
}
```

其中_PyObject_GetDictPtr 的代码就是 PyObject_GenericGetAttr 中根据 dictoffset 获得 dict 对象的那段代码。

### 12.5.2 再论 descriptor

在伪代码中，出现了“descriptor”，这个命名其实是有意为之的，目的是为了唤起读者对前面描述过的 descriptor 的回忆。前面我们看到，在 PyType_Ready 中，Python 虚拟机会填充 tp_dict，其中与操作名对应的是一个个 descriptor，那时我们看到的是

descriptor 这个概念在 Python 内部是如何实现的。现在我们将要剖析的是 descriptor 在 Python 的类机制中究竟会起到怎样的作用。

在 Python 虚拟机对 class 对象或 instance 对象进行属性访问时，descriptor 将对属性访问的行为产生重大的影响。一般而言，对于一个 Python 中的对象 obj，如果 obj.__class__对应的 class 对象中存在__get__、__set__和__delete__三种操作，那么 obj 就可称为 Python 一个 descriptor。在 slotdefs 中，我们会看到__get__、__set__、__delete__对应的操作：

```
[slotdefs in typeobject.c]
    ……
    TPSLOT("__get__", tp_descr_get, ……),
    TPSLOT("__set__", tp_descr_set, ……),
    TPSLOT("__delete__", tp_descr_set, ……),
    ……
```

前面我们看到了 PyWrapperDescrObject、PyMethodDescrObject 等对象，它们对应的 class 对象中分别为 tp_descr_get 设置了 wrapperdescr_get、method_get 等函数，所以它们绝对是当之无愧的 descriptor。

如果细分，那么 descriptor 还可分为如下两种：

- data descriptor ：type 中定义了__get__和__set__的 descriptor；
- non data descriptor ：type 中只定义了__get__的 descriptor。

在 Python 虚拟机访问 instance 对象的属性时，descriptor 的一个作用是影响 Python 虚拟机对属性的选择。从 PyObject_GenericGetAttr 的伪代码可以看出，Python 虚拟机会在 instance 对象自身的__dict__中寻找属性，也会在 instance 对象对应的 class 对象的 mro 列表中寻找属性，我们将前一种属性称为 instance 属性，而后一种属性称为 class 属性。

虽然 PyObject_GenericGetAttr 里对属性进行选择的算法比较复杂，但是从最终的效果上，我们可以总结出如下的两条规则：

- Python 虚拟机按照 instance 属性、class 属性的顺序选择属性，即 instance 属性优先于 class 属性；
- 如果在 class 属性中发现同名的 data descriptor，那么该 descriptor 会优先于 instance 属性被 Python 虚拟机选择。

这两条规则在对属性进行设置时仍然会被严格遵守，换句话说，如果执行“a.value = 1”，就算在 A 中发现一个名为“value”的 no data descriptor，那么还是会设置 a.__dict__[value] = 1，而不会设置 A 中已有的 value 属性。

当最终获得的属性是一个 descriptor 时，最奇妙的事发生了，Python 虚拟机不是简单地返回 descriptor，而是如伪代码所示的那样，调用 descriptor.__get__，将调用的结果返回。在下面的图 12-25 中，展示了 descriptor 对属性访问行为的影响。

```
>>> class A(list):
	def __get__(self, obj, cls):
		return 'A.__get__'


>>> class B(object):
	value = A()


>>> b = B()
>>> b.value
'A.__get__'
>>> s = b.value
>>> type(s)
<type 'str'>
```

descriptor改变返回值

```
>>> class A(list):
	def __get__(slef, obj, cls):
		return 'A.__get__'


>>> class B(object):
	value = A()


>>> b = B()
>>> b.value = 1
>>> b.__dict__['value']
1
>>> b.__class__.__dict__['value']
[]
```

instance属性优先于non data descriptor

```
>>> class A(list):
	def __get__(self, obj, cls):
		return 'A.__get__'
	def __set__(self, obj, value):
		print 'A.__set__'
		self.append(value)


>>> class B(object):
	value = A()


>>> b = B()
>>> b.value = 1
A.__set__
>>> b.__dict__['value']

Traceback (most recent call last):
  File "<pyshell#101>", line 1, in <module>
    b.__dict__['value']
KeyError: 'value'
>>> b.__class__.__dict__['value']
[1]
```

data descriptor优先于instance属性

图 12-25 descriptor 对属性访问行为的影响

有趣的是，如果我们在 class_0.py 中加入“print A.name”这样访问 class 对象的属性的表达式时，Python 虚拟机同样会为这个表达式编译出包含 LOAD_ATTR 指令的字节码序列，但不同的是，在 PyObject_GetAttr 中，最终会调用 type_getattro，而非 PyObject_GenericGetAttr。其实两者的算法都是类似的，上面所描述的所有结论都适合于 type_getattro。但是怎么理解 type_getattro 中的“instance 属性”呢？很简单，所有的 class 对象都是 metaclass 对象创建的，也就是 metaclass 的 instance，所以，对于 class 对象而言，“instance 属性”这个说法也是非常自然的，就是 class 对象自身的 tp_dict 中存储的属性。

前面我们说当访问的属性最终对应的是一个 descriptor 时，会调用其__get__方法，并将__get__的结果作为属性的值返回。其实这个说法不是完全正确的。仔细对比 type_getattro 和 PyObject_GenericGetAttr 的代码，我们会发现它们在如何对待 descriptor 的一个细节上存在着差异。在 PyObject_GenericGetAttr 中，如果查询到

的 descriptor 是 instance 属性，那么不会调用其__get__方法；而在 type_getattro 中，即使查询到的 descriptor 是“instance 属性”，也会调用其__get__方法。这样的区别可以用一句话来总结：**如果待访问的属性是一个 descriptor，若它存在于 class 对象的 tp_dict 中，会调用其__get__方法；若它存在于 instance 对象的 tp_dict 中，则不会调用其__get__方法**。在图 12-26 中给出了这条规则的一个例子。有兴趣的读者可以追踪一下 type_getattro 源码中的行为。

```
>>> class A(object):
        def __get__(self, obj, cls):
            return 'Python'


>>> class B(object):
        desc_in_class = A()


>>> B.desc_in_class
'Python'
>>> b = B()                        在class对象的tp_dict中的descriptor
>>> b.desc_in_class
'Python'
>>> b.desc_in_instance = A()
>>> b.desc_in_instance --> 在instance对象的tp_dict中的descriptor
<__main__.A object at 0x00DCDD70>
```

图 12-26 instance 对象和 class 对象中 descriptor 的不同行为

到这里，我们已经看到，descriptor 对属性访问的影响主要在两个方面：其一是对访问顺序的影响；其二是对访问结果的影响。第二种影响正是类的成员函数调用的关键。

## 12.5.3 函数变身

在前面讨论创建 <instance A> 时，我们看到，在 A.__dict__中，保存了一个与符号“f”对应的 PyFunctionObject 对象，所以在伪代码中的 descriptor 对应的就是一个 PyFunctionObject 对象。先抛开伪代码中确定最终返回值的过程不说，我们从另一个角度来看一看，假设 PyFunctionObject 作为 LOAD_ATTR 的最终结果，在 LOAD_ATTR 指令代码的最后被 SET_TOP 压入到了运行时栈中，那么会有什么后果呢？

在 A 的成员函数 f 的 def 语句中，我们分明看到一个 self 参数，self 在 Python 中是不是一个真正有效的参数呢？还是它仅仅是一个语法意义上的占位符而已？这一点可以从函数 g 中看到答案，在 g 中有这样的语句：self.value = aValue。这条语句毫无疑问地揭示了 self 确实应该是一个实实在在的对象，所以表面上看起来是无参函数的 f 其实是一个货真价实的带参函数。现在，问题来了，根据我们之前对函数机制的分析，Python 通常会将参数事先压入运行时栈中，但是从 class_0.py 中 a.f()语句编译后的指令序列中可以看到，Python 在获得了 a.f 对应的对象之后，没有进行任何如普通函数调用一样的参数入栈的动作，而是直接执行了 CALL_FUNCTION 指令。这里没有任何像参数的东西在栈

中，栈中只有一个我们认为是 PyFunctionObject 对象的 a.f 的返回结果，这个遗失的 self 参数究竟在什么地方呢？

好了，爱因斯坦说，为了相对论，我们必须抛弃绝对时空的观念，现在，为了能让 f 确实能得到一个参数，我们必须抛弃 PyFunctionObject 是返回结果的假设，它只能是另一种我们现在为止还不知道的全新的对象。由于是通过访问属性“f”得到的这个对象，所以一个合理的假设是，在这个对象中，包含了“f”对应的那个 PyFunctionObject 对象；另一个更合理的假设是，在这个对象中，还包含了函数 f 的参数：self。

为了能看清这个全新的神秘对象究竟是谁，让我们回到伪代码中来。在第 11 章对函数机制的剖析中，我们对 PyFunctionObject 似乎已经了如指掌了。但那时，我们不经意间放过了 PyFunctionObject 对象对应的 class 对象——PyFunction_Type。在这个 PyFunction_Type 中，其实隐藏着一个惊天大秘密。观察 PyFunctino_Type，我们会发现与“__get__”对应的 tp_descr_get 被设置成了&func_descr_get。这意味着我们这里得到的 A.f 实际上是一个 descriptor。由于 PyFunction_Type 中并没有设置 tp_descr_set，所以 A.f 是一个 non data descriptor。此外，由于在 a.__dict__中没有符号“f”存在，所以根据伪代码中的算法，a.f 的返回值将被 descriptor 改变，其结果将是 A.f.__get__，也就是 func_descr_get(A.f, a, A)：

```
[funcobject.c]
/* Bind a function to an object */
static PyObject* func_descr_get(PyObject *func, PyObject *obj, PyObject
 *type)
{
    if (obj == Py_None)
        obj = NULL;
    return PyMethod_New(func, obj, type);
}
```

func_descr_get 将 A.f 对应的 PyFunctionObject 进行了一番包装，通过 PyMethod_New，Python 虚拟机在 PyFunctionObject 的基础上创建了一个新的对象，到 PyMethod_New 中一看，那个神秘的对象现身了：

```
[classobject.c]
PyObject* PyMethod_New(PyObject *func, PyObject *self, PyObject *class)
{
    register PyMethodObject *im;
    im = free_list;
    if (im != NULL) {
        //使用缓冲池
        free_list = (PyMethodObject *)(im->im_self);
        PyObject_INIT(im, &PyMethod_Type);
    }
    else {
        //不使用缓冲池，直接创建 PyMethodObject 对象
        im = PyObject_GC_New(PyMethodObject, &PyMethod_Type);
```

```
    }
    im->im_weakreflist = NULL;
    im->im_func = func;
    //喏，这里就是“self”啦~~
    im->im_self = self;
    im->im_class = class;
    _PyObject_GC_TRACK(im);
    return (PyObject *)im;
}
```

一切真相大白，原来那个神秘的对象就是 PyMethodObject 对象。看到 free_list 这样熟悉的字眼，现在我们立即可以判断出，在 PyMethodObject 的实现和管理中，Python 采用了缓冲池的技术。现在来看看这个 PyMethodObject：

```
[classobject.h]
typedef struct {
    PyObject_HEAD
    PyObject *im_func;   //可调用的 PyFunctionObject 对象
    PyObject *im_self;   //用于成员函数调用的 self 参数，instance 对象(a)
    PyObject *im_class;  //class 对象(A)
    PyObject *im_weakreflist;
} PyMethodObject;
```

在 PyMethod_New 中，分别将 im_func，im_self，im_class 设置了不同的值，对应的就是“f”对应的 PyFunctionObject 对象，“a”对应的 instance 对象，以及“A”对应的 class 对象。

在 Python 中，将 PyFunctionObject 对象和一个 instance 对象通过 PyMethodObject 对象结合在一起的过程就称为成员函数的绑定。下面的图 12-27 清晰地展示了在访问属性时，发生的函数绑定的结果：

```
>>> class A(object):
        def f(self):
                pass


>>> a = A()
>>> a.__class__.__dict__['f']
<function f at 0x00DB04F0>
>>> a.f
<bound method A.f of <__main__.A object at 0x00DC61B0>>
```

图 12-27 函数绑定的结果

## 12.5.4 无参函数的调用

在 LOAD_ATTR 指令之后，指令“37 CALL_FUNCTION 0”开始了函数调用的动作，之前我们研究过了对于 PyFunctionObject 对象的调用，而对于 PyMethodObject 对象，情况有些不同，见代码清单 12-6：

代码清单 12-6

```
[ceval.c]
static PyObject* call_function(PyObject ***pp_stack, int oparg)
{
    int na = oparg & 0xff;
    int nk = (oparg>>8) & 0xff;
    int n = na + 2 * nk;
    PyObject **pfunc = (*pp_stack) - n - 1;
    PyObject *func = *pfunc;
    PyObject *x, *w;

    if (PyCFunction_Check(func) && nk == 0) {
        ......
    } else {
        //[1]: 从 PyMethodObject 对象中抽取 PyFunctionObject 对象和 self 参数
        if (PyMethod_Check(func) && PyMethod_GET_SELF(func) != NULL) {
            PyObject *self = PyMethod_GET_SELF(func);
            func = PyMethod_GET_FUNCTION(func);
            //[2]: self 参数入栈，调整参数信息变量
            *pfunc = self;
            na++;
            n++;
        }
        if (PyFunction_Check(func))
            x = fast_function(func, pp_stack, n, na, nk);
        else
            x = do_call(func, pp_stack, na, nk);
    }
    ......
    return x;
}
```

显然，调用成员函数 f 时，显式传入的参数个数为 0，也就是说，调用 f 时，Python 虚拟机没有进行参数入栈的动作。而 f 显然又是需要一个参数的函数，其参数为 self，正是在 call_function 中，Python 虚拟机为 PyMethodObject 进行了一些参数处理的动作。

当 Python 虚拟机执行 a.f() 时，在 call_function 中，代码清单 12-6 中[1]处的判断会成立，其中的 PyMethod_GET_SELF 被定义为：

```
[classobject.h]
#define PyMethod_GET_SELF(meth) (((PyMethodObject *)meth) -> im_self)
```

在 call_function 中，func 变量指向一个 PyMethodObject 对象，而在清单 12-6 中[1]处的判断成立后，在 if 分支中又会将 PyMethodObject 对象中的 PyFunctionObject 对象和 instance 对象分别提取出来。在 if 分支中有一处最重要的代码：

```
PyObject *self = PyMethod_GET_SELF(func);
. . . . . .
*pfunc = self;
```

还记得我们在分析函数机制时看到的 pfunc 的意义吗？它指向的位置正是运行时栈

中存放 PyMethodObject 对象的位置。那么在这个本来属于 PyMethodObject 的地方存放 instance 对象究竟有什么作用呢？在这里，Python 虚拟机以另一种方式完成了函数参数入栈的动作，而这个本来属于 PyMethodObject 对象的内存空间现在被用作了函数 f 的 self 参数的容身之处。

我们先来看看图 12-28 所示的运行 call_function 时运行时栈的变化情况：

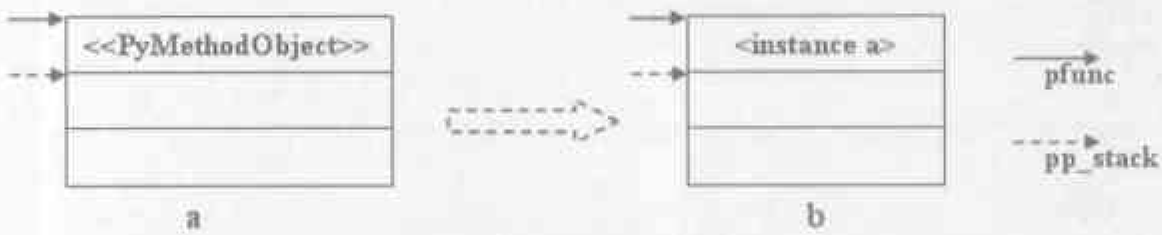



图 12-28 设置 self 参数

图 12-28 中 a 是设置 pfunc 之前的运行时栈，b 表示设置了 pfunc 之后的运行时栈。这里似乎看不出什么玄妙之处，别急，我们继续前进。

在 call_function 中，接着还会通过 PyMethod_GET_FUNCTION 将 PyMethodObject 对象中的 PyFunctionObject 对象提取出来。随后在 call_function 中的[2]处（见代码清单 12-6），Python 虚拟机完成了 self 参数的入栈，同时还调整了维护着参数信息的 na 和 n，回忆一下第 11 章对 Python 函数机制的分析，我们可以发现，调整后的结果意味着函数会获得一个位置参数，看一看 A 中 f 的 def 语句，没错，self 正是一个位置参数。

由于 func 在 if 分支之后指向了 PyFunctionObject 对象，所以接下来 Python 执行引擎将进入 fast_function。到了这里，我们已经可以豁然贯通了，剩下的动作将和我们之前所分析过的带参函数的调用一致。实际上 a.f 的调用实质上就是一个带 1 个位置参数的一般函数的调用。而在 fast_function，在图 12-27 中作为 self 参数的 <instance a> 被 Python 虚拟机压入到了运行时栈中。从第 11 章的分析我们可以知道，在 fast_function 中，由于 a.f 仅仅是一个带位置参数的函数，所以 Python 执行引擎将进入快速通道，在快速通道中，运行时栈中的这个 instance 对象会被拷贝到新的 PyFrameObject 对象的 f_localsplus 中。还记得我们曾经翻来覆去剖析过的 fast_function 吗？

**代码清单 12-7**

```
[ceval.c]
PyObject*
fast_function(PyObject *func, PyObject ***pp_stack, int n, int na, int nk)
{
    ......
    if (argdefs == NULL && co->co_argcount == n && nk==0 &&
        co->co_flags == (CO_OPTIMIZED | CO_NEWLOCALS | CO_NOFREE)) {
        ......//创建新的 PyFrameObject 对象 f
        fastlocals = f->f_localsplus;
        stack = (*pp_stack) - n; //[1] : 获得栈顶指针
```

```
        for (i = 0; i < n; i++) {
            fastlocals[i] = *stack++; //[2]: 拷贝 self 参数到 f->f_localsplus
        }
        ......
    }
    ......
}
```

在调用 fast_function 时，参数的数量 n 已经由执行 CALL_FUNCTION 时的 0 变为了 1，所以代码清单 12-7 中[1]处的 stack 指向的位置就和图 12-27 中的 pfunc 指向的位置是一致的了。为了要在代码清单 12-7 的[2]处将<instance a>作为参数拷贝到函数的参数区 fastlocals 中，必须将它放置到栈顶，也就是以前 PyMethodObject 对象所在的位置上。这就是前面在 call_function 中那个神秘的赋值操作的原因。

## 12.5.5 带参函数的调用

Python 虚拟机对类中带参的成员函数的调用，其原理和流程都与无参函数的调用是一致的，我们来看看调用函数 g 的字节码序列：

```
[PyCodeObject for class_0.py]
a.g(10)
41    LOAD_NAME         2 (a)
44    LOAD_ATTR         4 (g)
47    LOAD_CONST        2 (10)
50    CALL_FUNCTION     1
53    POP_TOP
```

可以看到，和调用成员函数 f 的指令序列几乎完全一致，只是多出了一个"LOAD_CONST 2"指令。对这个指令应该不陌生了，在分析函数机制的时候，我们看到它是用来将函数所需的参数压入到运行时栈中的。这就是带参函数与无参函数的唯一不同之处，想一想无参函数的调用流程，带参函数的调用流程也迎刃而解了。

对于 g，真正有趣的地方在于考察函数的实现代码，从而可以看到对那个作为 self 参数的 instance 对象的使用：

```
[PyCodeObject for function g]
    def g(self, aValue):
        self.value = aValue
        0   LOAD_FAST     1 (aValue)
        3   LOAD_FAST     0 (self)
        6   STORE_ATTR    0 (value)

        print self.value
        9    LOAD_FAST    0 (self)
        12   LOAD_ATTR    0 (value)
        15   PRINT_ITEM
        16   PRINT_NEWLINE
```

显然，其中的 LOAD_FAST、LOAD_ATTR、STORE_ATTR 这些字节码指令都涉及到了作为 self 参数的 instance 对象。有兴趣的读者可以分析一下 STORE_ATTR 的代码，可以发现其中也有类似于 LOAD_ATTR 中 PyObject_GenericGetAttr 的属性访问算法。

其实到了这里，我们可以在更高的层次俯视一下 Python 的运行模型了，最核心的模型其实非常简单，可以简化为两条规则：

- 在某个名字空间中寻找符号对应的对象；
- 对从名字空间中得到的对象进行某些操作。

抛开面向对象花里胡哨的外表，其实我们会发现，class 对象其实就是一个名字空间，instance 对象也是一个名字空间，不过这些名字空间通过一些特殊的规则联结在一起，使得符号的搜索过程变得复杂，从而实现了面向对象这种编程模式，如此而已。

## 12.5.6 Bound Method 和 Unbound Method

在 Python 中，当对作为属性的函数进行引用时，会有两种形式，一种称为 Bound Method，这种形式是通过类的实例对象进行属性引用，就像我们之前花了大力气讨论的 a.f 这样的形式；而另一种则是通过类进行属性引用，称为 Unbound Method，形式如 A.f，当然，对 Bound Method 的调用和对 Unbound Method 的调用其形式是不同的，其原因可以追溯到 LOAD_ATTR 中。我们先来看一段使用 Unbound Method 的 Python 代码——class_1.py：

```
[class_1.py]
class A:
    def g(self, value):
        self.value = value
        print self.value
a = A()
A.g(a, 10)
31   LOAD_NAME     1 (A)    //<class A>入栈
34   LOAD_ATTR     3 (g)    //获得 A 中的属性 g
37   LOAD_NAME     2 (a)    //参数<instance a>入栈
40   LOAD_CONST    2 (10)   //参数<int 10>入栈
43   CALL_FUNCTION   2      //调用函数 g
```

其中的关键就在于“34 LOAD_ATTR 3”指令，在 12.5.2 节中，我们已经解释过，这里的 LOAD_ATTR 指令最终会调用 type_getattro。在 type_getattro 中，会在<class A>的 tp_dict 中发现“g”对应的 PyFunctionObject，同样，因为它是一个 descriptor，因此也会调用其__get__函数进行转变。在之前剖析 a.f 时，我们看到这个转变是通过 func_descr_get(A.f, a, A)完成的，这里对“g”的转变则是通过 func_descr_get(A.g, NULL, A)完成。因此，虽然 A.g 也得到了一个 PyMethodObject，但是其中的 im_self 却是 NULL。

在 Python 中，Bound Method 与 Unbound Method 的本质区别就在于 PyFunctionObject 有没有与 instance 对象绑定在 PyMethodObject 中，Bound Method 完成了绑定动作，而 Unbound Method 没有完成绑定动作。

所以，在对 Unbound Method 进行调用时，我们必须显式地提供一个 instance 对象作为函数的第一个位置参数，因为 g 无论如何都需要一个 self 参数。所以才会有 A.g(a, 10) 这样的调用形式。而无论是对 Unbound Method 进行调用，还是对 Bound Method 进行调用，Python 虚拟机的动作在本质上都是一样的，都是调用带位置参数的一般函数，区别只是在于：当调用 Bound Method 时，Python 虚拟机帮我们完成了 PyFunctionObject 对象和 instance 对象的绑定，instance 对象将自动成为 self 参数；而调用 Unbound Method 时，没有这个绑定，我们需要自己传入 self 参数。

下面的图 12-29 清晰地展示出了 Bound Method 与 Unbound Method 的不同：

```
>>> class A(object):
        def f(self):
            pass


>>> a = A()
>>> bound = a.f
>>> unbound = A.f
>>>
>>> bound
<bound method A.f of <__main__.A object at 0x00DCCC10>>
>>> unbound
<unbound method A.f>
>>>
>>> bound.im_self
<__main__.A object at 0x00DCCC10>
>>> unbound.im_self
>>>
>>>
```

图 12-29 bound method 和 unbound method

在我们希望输出 Unbound Method 对象的 im_self 域时，没有任何东西输出，因为这个域根本就是个 NULL。

对于成员函数的调用，或者说是对于成员函数的绑定过程，有一点值得注意的是，每一次函数调用都会激发一次绑定过程。其原因在于，每次进行属性引用时，都会重新获得属性对应的 PyFunctionObject(descritpro)，进而创建新的 PyMethodObject 对象。这一点的开销实在是有些大。表 12-3 展示了两种不同函数调用方式的绑定次数（我们修改了 PyMethod_New 的代码，使其在创建名为 f 的 PyMethodObject 对象时输出调用次数）：

表 12-3 不同函数调用方式对应的绑定次数

| 文件名 | bound_1.py | bound_2.py |
|---|---|---|
| 代码 | `class A:`<br>`    def f(self):`<br>`        pass`<br>`a = A()`<br>`for i in range(100):`<br>`    a.f()` | `class A:`<br>`    def f(self):`<br>`        pass`<br>`a = A()`<br>`func = a.f`<br>`for i in range(100)`<br>`func(10)` |
| 输出结果 | 函数绑定 100 次 | 函数绑定 1 次 |

在 bound_1.py 中，一共会进行 100 次的属性访问和函数绑定操作，而在 bound_2.py 中却只会进行一次属性访问和函数绑定的操作。

## 12.6 千变万化的 descriptor

当我们调用 instance 对象的函数时，最关键的一个动作就是从 PyFunctionObject 对象向 PyMethodObject 对象的转变，而这个关键的转变被 Python 中的 descriptor 概念很自然地融入到 Python 的类机制中。当我们访问对象中的属性时，由于 descriptor 的存在，这种转换就自然而然地发生了。将这种 descriptor 的思想推而广之，其实在访问属性时，我们不光能实现从 PyFunctionObject 到 PyMethodObject 对象的转变，实际上我们可以做任何事情。在 Python 内部，也存在着各种各样的 descriptor，这些 descriptor 的存在给 Python 的类机制赋予了强大的力量。这一节，我们就来看看 Python 是如何使用 descriptor 实现 static method 的。

```
[class_2.py]
class A(object):
    def g(value):
        print value
    g = staticmethod(g)
```

在 3.1.1 节描述的为 A 创建动态元信息的过程中，Python 虚拟机首先会执行一个 def 语句，将符号“g”和一个 PyFunctionObject 对象关联起来，但随后的 g = staticmethod(g) 则会将“g”与一个 staticmethod 对象关联起来，从而将属性“g”改造成一个 static method。这条语句对应的字节码指令如下：

```
[PyCodeObject for class A]
def g(value):
    print value
g = staticmethod(g)
15 LOAD_NAME              3 (staticmethod)
18 LOAD_NAME              2 (g)
21 CALL_FUNCTION          1
24 STORE_NAME             2 (g)
```

Python 虚拟机在执行“15 LOAD_NAME 3”指令时，会从 builtin 名字空间中获得一个与符号“staticmethod”对应的对象，这个对象在 Python 启动并进行初始化时设置（初始化过程将在后面的章节中详细剖析），从图 12-30 可以看到，它其实是一个 class 对象——<type staticmethod>：

```
>>> staticmethod
<type 'staticmethod'>
>>>
```

图 12-30 builtin 名字空间中的“staticmethod”

所以，执行 staticmethod(g)的过程就是一个从 class 对象创建 instance 对象的过程，从本章前面的描述我们可以知道，最终将调用 PyObject_GenericAlloc 申请一段内存，内存空间的大小由 staticmethod 结构体决定：

```
[funcobject.c]
typedef struct {
    PyObject_HEAD
    PyObject *sm_callable;
} staticmethod;
```

在申请完内存之后，Python 虚拟机还会调用__init__进行初始化的动作，<type staticmethod>在 Python 内部对应的是 PyStaticMethod_Type，而其中的 tp_init 设置为 sm_init：

```
[funcobject.c]
static int sm_init(PyObject *self, PyObject *args, PyObject *kwds)
{
    staticmethod *sm = (staticmethod *)self;
    PyObject *callable;

    PyArg_UnpackTuple(args, "staticmethod", 1, 1, &callable))
    sm->sm_callable = callable;
    return 0;
}
```

非常清晰，在初始化时，作为参数的原来“g”对应的那个 PyFunctionObject 被赋给了 staticmethod 对象中的 sm_callable。最后 Python 虚拟机通过指令“24 STORE_NAME 2”将符号“g”和这个 staticmethod 对象关联了起来。

再仔细地考察 PyStaticMethod_Type，发现这里创建的 staticmethod 对象实际上也是一个 descriptor，因为在 PyStaticMethod_Type 中，tp_descr_get 指向了 sm_descr_get。

```
[funcobject.c]
static PyObject *sm_descr_get(PyObject *self, PyObject *obj, PyObject
*type)
{
    staticmethod *sm = (staticmethod *)self;
    return sm->sm_callable;
}
```

当我们访问属性“g”时，不论是通过 instance 对象访问(a.g)，还是通过 class 对象访问(A.g)，由于“g”是一个位于 class 对象<class A>的 tp_dict 中的 descriptor，所以会调用其__get__操作（sm_descr_get），直截了当地返回其中保存的最开始与“g”对应的那个 PyFunctionObject 对象。

Python 中的 static method 只是 descriptor 应用的一个例子，还有其他很多特性，比如 class method、property 都是应用 descriptor 的例子，而且，到目前为止，我们也没有再剖析初识 descriptor 时遇见的那个 PyWrapperDescrObject，有兴趣的读者可以自行分析一下。

# 第3部分

# Python高级话题

CHAPTER

第 13 章

# Python 运行环境初始化

我们现在已经完成了对 Python 的核心——字节码虚拟机——的剖析工作，然而对于完整地理解 Python 运行时的行为还不够，还有一部分内容被遮在了大幕后边。在这一章，我们将回到时间的起点，从 Python 应用程序被执行开始，一步一步紧紧跟随 Python 的踪迹，完整地展示 Python 在启动之初的所有动作。当我们跟随 Python 完成所有的初始化动作之后，也就能对 Python 执行引擎执行字节码指令时的整个运行环境了如指掌了。

## 13.1 线程环境初始化

### 13.1.1 线程模型回顾

Python 启动之后，真正有意义的初始化动作是从 Py_Initialize 开始的。在 Py_Initialize 中，仅有一个函数被调用，即函数 Py_InitializeEx。尽管在 Py_Initialize 之前，Python 已经做了很多繁琐的工作，但是这些工作对于理解 Python 的运行环境并没有太多的意义，所以我们对 Python 运行时环境初始化的剖析也从 Py_InitializeEx 开始。

在 Py_InitializeEx 中，所完成的一个重要的工作就是加载多个基础的 module，比如__builtin__，sys 等，同时也会完成 Python 类型系统的初始化和异常系统的初始化，当然还有其他许多工作，在本章以后的描述中我们将深入考察这些工作。

但是在进入对 Python 初始化流程的跟踪之前，我们需要先复习一下 Python 的运行模型，或者说线程模型，在第 8 章中，我们曾详细地介绍了这个线程模型。在此，我们仅列出两个关键的数据结果以及对 Python 运行模型的图示。

```
[pystate.h]
typedef struct _is {
    struct _is *next;
```

```
    struct _ts *tstate_head; //模拟进程环境中的线程集合

    PyObject *modules;
    PyObject *sysdict;
    PyObject *builtins;
    ......
} PyInterpreterState;

typedef struct _ts {
    struct _ts *next;
    PyInterpreterState *interp;
    struct _frame *frame; //模拟线程中的函数调用堆栈
    int recursion_depth;
    ......
    PyObject *dict;
    ......
    long thread_id;
} PyThreadState;
```

其中，PyInterpreterState 是对进程的模拟，而 PyThreadState 是对线程的模拟。图 13-1 则展示了 Python 虚拟机运行期间某个时刻整个的运行环境。

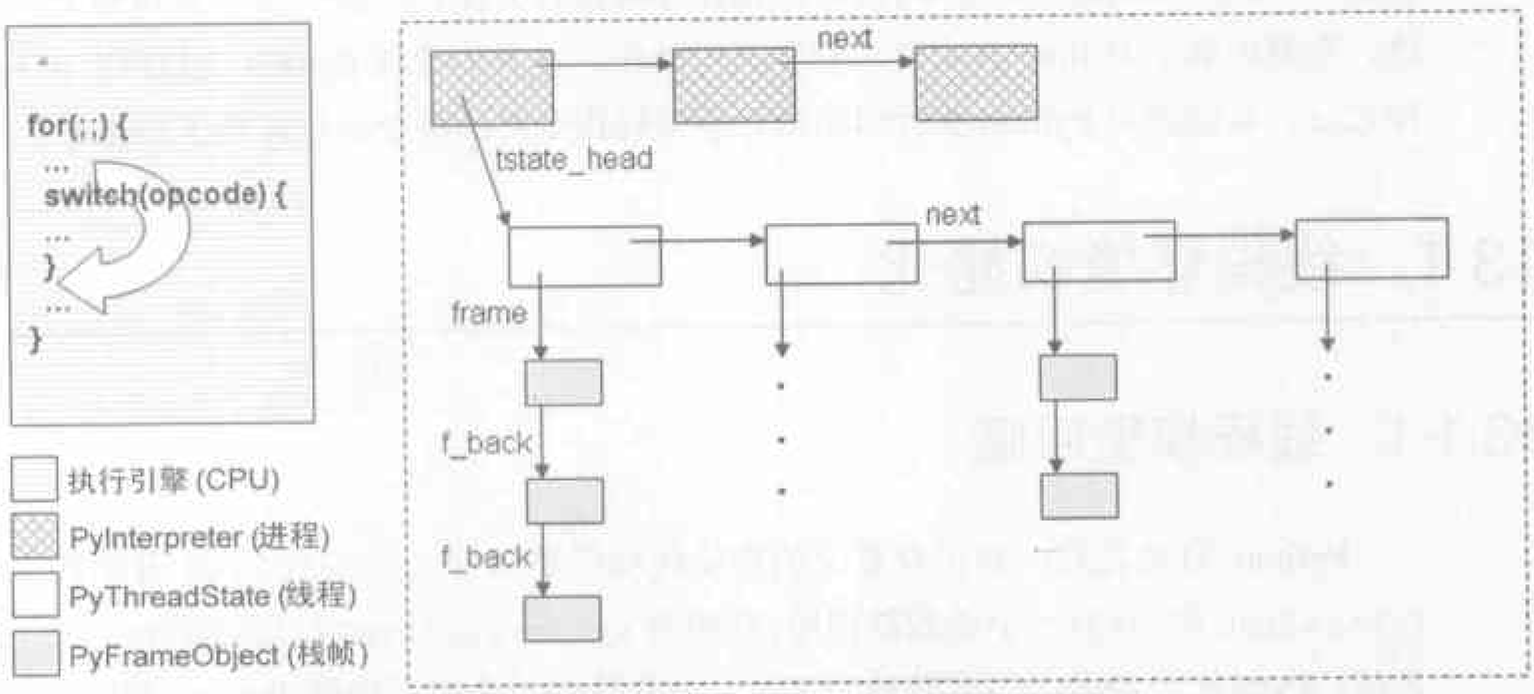



图 13-1 某个时刻 Python 运行时的整个环境

## 13.1.2 初始化线程环境

在 Win32 平台上，当执行一个可执行文件时，操作系统首先会创建一个进程内核对象。同样，在 Python 中也是如此，在 Py_InitializeEx 的开始处，Python 会首先调用 PyInterpreterState_New 创建一个崭新的 PyInterpreterState 对象。

```
[pystate.c]
static PyInterpreterState *interp_head = NULL;

PyInterpreterState* PyInterpreterState_New(void)
```

```
{
    PyInterpreterState *interp = malloc(sizeof(PyInterpreterState));
    if (interp != NULL) {
        HEAD_INIT();
        interp->modules = NULL;
        interp->sysdict = NULL;
        interp->builtins = NULL;
        interp->tstate_head = NULL;
        interp->codec_search_path = NULL;
        interp->codec_search_cache = NULL;
        interp->codec_error_registry = NULL;
        HEAD_LOCK();
        interp->next = interp_head;
        interp_head = interp;
        HEAD_UNLOCK();
    }

    return interp;
}
```

在 Python 的运行时环境中，有一个全局的管理 PyInterpreterState 对象链表的东西：interp_head。从这个名字以及 PyInterpreterState 结构体中的 next 指针看，我们可以断定在 Python 运行的时候，可能会有一组 PyInterpreterState 对象通过 next 指针形成一个链表结构，而这个表头就是 interp_head。这其实就是如图 13-1 所示的 Python 对操作系统上多进程的模拟，对于多进程，这里我们不过多深入。

在 PyInterpreterState_New 函数完成之后，我们得到了如图 13-2 所示的解释器状态对象。

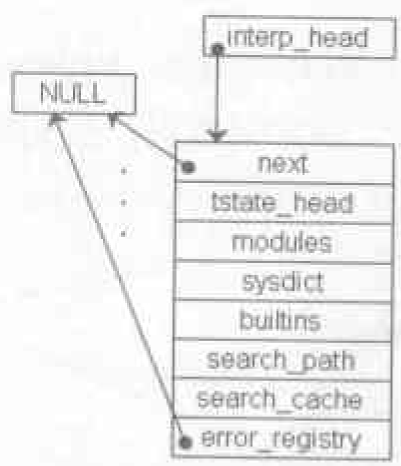



图 13-2　新建的 PyInterpreterState 对象

在创建了 PyInterpreterState（进程状态）对象之后，Python 会立即再接再厉，调用 PyThreadState_New 创建一个全新的 PyThreadState（线程状态）对象（见代码清单 13-1）。

代码清单 13-1

```
[pystate.c]
PyThreadState* PyThreadState_New(PyInterpreterState *interp)
{
```

```
    PyThreadState *tstate = (PyThreadState *)malloc(sizeof(PyThreadState));
    //[1]: 设置获得线程中函数调用栈的操作
    if (_PyThreadState_GetFrame == NULL)
        _PyThreadState_GetFrame = threadstate_getframe;

    if (tstate != NULL) {
    //[2]: 在 PyThreadState 对象中关联 PyInterpreterState 对象
        tstate->interp = interp;
        tstate->frame = NULL;
        tstate->thread_id = PyThread_get_thread_ident();
        ……
        HEAD_LOCK();
        tstate->next = interp->tstate_head;
        //[3]: 在 PyInterpreterState 对象中关联 PyThreadState 对象
        interp->tstate_head = tstate;
        HEAD_UNLOCK();
    }
    return tstate;
}
```

与 PyInterpreterState_New 相同，PyThreadState_New 申请内存，创建 PyThreadState 对象，并对其中各个域进行初始化。我们注意到，在 PyThreadState 结构体中，也存在着一个 next 指针，肯定在 Python 运行的某个时刻，会如图 13-1 所示的那样，存在一个 PyThreadState 对象的列表，用脚趾头我们也能想到这肯定和 Python 中多线程的实现有关。

在代码清单 13-1 的[1]处，Python 设置了从线程中获得函数调用栈的方法，所谓的函数调用栈也就是 PyFrameObject 对象链表。

注意，在代码清单 13-1 的[2]和[3]处，Python 到目前为止创建的仅有的两个对象建立起了联系，对应到 Win32 上，我们可以说，在[2]和[3]处，进程和线程之间建立起了联系。建立联系后的 PyThreadState 对象和 PyInterpreterState 对象如图 13-3 所示。

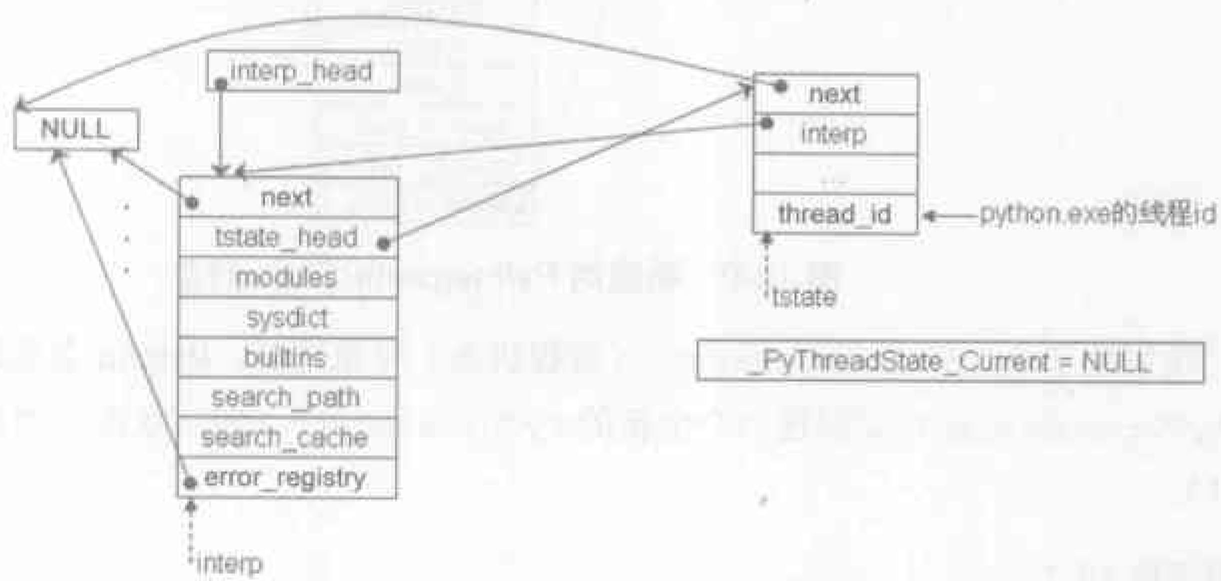



图 13-3 在 PyInterpreterState 与 PyThreadState 之间建立联系

注意，在图 13-3 中，虚线箭头表示在函数 PyThreadState_New 中 interp 和 tstate 两个变量所指向的对象。

在 PyInterpreterState 对象和 PyThreadState 对象建立了这样的联系之后，我们就能够很容易地在 PyInterpreterState 对象和 PyThreadState 对象之间穿梭往返。在 Python 的运行时环境中，有一个全局变量_PyThreadState_Current，这个变量维护着当前活动的线程，更准确地说是当前活动线程对应的 PyThreadState 对象，初始时，该变量为 NULL，如图 13-3 所示。在创建了 Python 启动后的第一个 PyThreadState 对象之后，会以该 PyThreadState 对象调用 PyThreadState_Swap 函数来设置这个全局变量：

```
[pystate.c]
PyThreadState* PyThreadState_Swap(PyThreadState *new)
{
    PyThreadState *old = _PyThreadState_Current;
    _PyThreadState_Current = new;
    return old;
}
```

接着，Python 的初始化动作开始转向 Python 类型系统的初始化，这个转折是在 Py_InitializeEx 中调用_Py_ReadyTypes 时开始的。类型系统的初始化是一套相当繁复的动作，在介绍 Python 的类机制时，我们已经详细剖析了。

随后，在 Py_InitializeEx 中，接下来的动作是调用_PyFrame_Init 来设置全局变量 builtin_object：

```
[frameobject.c]
static PyObject *builtin_object;

int _PyFrame_Init()
{
    builtin_object = PyString_InternFromString("__builtins__");
    return (builtin_object != NULL);
}
```

这个内容为"__builtins__"的 PyStringObject 对象 builtin_object 在 PyFrame_New 创建一个新的 PyFrameObject 对象时将发挥作用，这里我们暂时不去管它。

在此之后，Python 还会初始化一些其他边边角角的东西，比如在_PyInt_Init 中初始化我们在剖析 Python 中的整数对象时看到的那个庞大的整数对象系统。当然，这里我们就不去深究这些初始化了，有兴趣的读者可以收拾兵马，杀将进去。

好了，到这里，我们的 Py_InitializeEx 有了一个阶段性的成果。我们创建了代表进程和线程概念的 PyInterpreterState 对象和 PyThreadState 对象，并且在它们之间建立了联系。接下来，Py_InitializeEx 将进入另一个相对独立的环节：设置系统 module。

## 13.2 系统 module 初始化

我们都有这样的经历，在交互模式下启动 Python 程序之后，敲入“dir()”，会显示一个 list 的内容，我们这里不管它是如何运作的，输出的信息又是什么，单说这个调用本身，就大有内容可挖。我们已经知道，Python 要执行 dir()，必定是在某个名字空间中寻找到了符号“dir”所对应的对象，更进一步，我们甚至知道，这个对象一定是个 callable 的对象。不过我们现在仅仅考虑符号“dir”的存在性，符号“dir”的存在性意味着在 Python 启动之后，虽然我们没有进行任何操作，但是 Python 已经创建了某个名字空间，在这个名字空间中，存在着符号“dir”。这个名字空间中的符号和值来自于系统 module，而这些系统 module 正是在 Py_InitializeEx 中设置的。其中第一个被 Python 创建的 module 是__builtin__ module。

### 13.2.1 创建__builtin__ module

在 Py_InitializeEx 中，当 Python 创建了 PyInterpreterState 和 PyThreadState 对象之后，就会开始通过_PyBuiltin_Init 来设置系统的__builtin__ module 了。

```
[pythonrun.c]
void Py_InitializeEx(int install_sigs)
{
    ......
    interp->modules = PyDict_New();
    bimod = _PyBuiltin_Init();
    ......
}
```

在调用_PyBuiltin_Init 之前，Python 最终会将 interp->modules 创建为一个 PyDictObject 对象，这个对象将维护系统所有的 module，这也符合我们在第 8 章中提到的，PyInterpreterState 对象中维护着所有的 PyThreadState 对象共享的资源。这一点我们将在接下来的_PyBuiltin_Init 中清晰地看到（见代码清单 13-2）。

**代码清单 13-2**

```
[bltinmodule.c]
PyObject* _PyBuiltin_Init(void)
{
    PyObject *mod, *dict, *debug;
    //[1]: 创建并设置__builtin__ module
    mod = Py_InitModule4("__builtin__", builtin_methods,
              builtin_doc, (PyObject *)NULL,
              PYTHON_API_VERSION);

    //[2]: 将所有 Python 内建类型加入到__builtin__ module 中
    dict = PyModule_GetDict(mod);
```

```
#define SETBUILTIN(NAME, OBJECT) \
    if (PyDict_SetItemString(dict, NAME, (PyObject *)OBJECT) < 0)    \
        return NULL;

    SETBUILTIN("None",       Py_None);
    ......
    SETBUILTIN("dict",       &PyDict_Type);
    ......
    SETBUILTIN("int",        &PyInt_Type);
    SETBUILTIN("list",       &PyList_Type);
    ......
    return mod;
#undef SETBUILTIN
}
```

整个_PyBuiltin_Init 函数的功能就是设置好__builtin__ module，作为一个Pythoner，对于这个家伙，应该不会陌生吧☺。_PyBuiltin_Init 通过两个步骤来完成对__builtin__的设置：

代码清单 13-2 的[1]和[2]分别有如下含义：

[1] 创建 PyModuleObject 对象，在 Python 中，module 正是通过这个对象来实现的。

[2] 设置 module，将 Python 中所有的类型对象全塞到新创建的__builtin__ module 中。

其实在第一步中，Python 就已经完成了大部分设置__builtin__ module 的工作，这部分工作是通过 Py_InitModule4 来完成的（见代码清单 13-3）。

**代码清单 13-3**

```
[modsupport.c]
PyObject* Py_InitModule4(const char *name, PyMethodDef *methods, char *doc,
           PyObject *passthrough, int module_api_version)
{
    PyObject *m, *d, *v, *n;
    PyMethodDef *ml;
    ......
    //[2]: 创建module对象
    if ((m = PyImport_AddModule(name)) == NULL)
        return NULL;

    //[3]: 设置module中的(符号, 值)对应关系
    d = PyModule_GetDict(m);
    if (methods != NULL) {
        n = PyString_FromString(name);
        //遍历methods指定的module对象中应包含的操作集合
        for (ml = methods; ml->ml_name != NULL; ml++) {
            if ((ml->ml_flags & METH_CLASS) || (ml->ml_flags & METH_STATIC)) {
                PyErr_SetString(PyExc_ValueError,
                        "module functions cannot set"
                        " METH_CLASS or METH_STATIC");
                return NULL;
            }
            v = PyCFunction_NewEx(ml, passthrough, n);
```

```
            PyDict_SetItemString(d, ml->ml_name, v)
        }
    }
    if (doc != NULL) {
        v = PyString_FromString(doc);
        PyDict_SetItemString(d, "__doc__", v);
    }
    return m;
}
```

Py_InitModule4 的函数参数对于理解这个家伙的行为非常有帮助，所以我们将各个参数的意义在下面列出。

- name：module 的名称，在这里，是"__builtin__"。
- methods：该 module 中所包含的函数的集合，在这里，是 builtin_methods。
- doc：module 的文档，在这里，是 builtin_doc。
- passthrough：这个参数在 Python 2.5 中并没有使用，所以你可以看到它是个 NULL。
- module_api_version：Python 内部使用的 version 值，用于比较（参考 Py_InitModule4 完整代码）。

我们看到，在 Py_InitModule4 中，Python 的行为可以分为相对独立的两个部分，一个是创建 module 对象本身，另一个是将（符号，值）对应关系放置到所创建的 module 中。

### 13.2.1.1 创建 module 对象

在函数 Py_InitModule4 中，代码清单 13-4 的[1]处的 PyImport_AddModule 创建了 module 对象本身。

**代码清单 13-4**

```
[import.c]
PyObject *
PyImport_AddModule(char *name)
{
    //[1]: 获得 Python 维护的 module 集合
    PyObject *modules = PyImport_GetModuleDict();
    PyObject *m;

    //[2]: 若module 集合中没有名为name 的module 对象，则创建之；否则，直接返回 module 对象
    if ((m = PyDict_GetItemString(modules, name)) != NULL &&
        PyModule_Check(m))
        return m;
    m = PyModule_New(name);

    //[3]: 将新创建的 module 对象放入 Python 的全局 module 集合中
    PyDict_SetItemString(modules, name, m);
    return m;
}
```

Python 内部维护了一个存放所有加载到内存中的 module 的集合，在这个集合中，存放着所有的（module 名，module 对象）这样的对应关系，这个集合其实就是我们在之前的 13.2.1 节中看到的 Py_InitializeEx 中出现的 interp->modules，从那里可以看到，它确实就是一个 PyDictObject 对象。对应到 Python 一级，这个全局的 modue 集合就是 sys.modules。

在 Python 创建一个新的 module 对象之前，会先到这个 module 集合中查看是否已经存在同名的 module。Python 通过代码清单 13-4 的[1]处的 PyImport_GetModuleDict 获得这个 PyInterpreterState 对象中的 module 集合。

```
[pystate.h]
#define PyThreadState_GET() (_PyThreadState_Current)

[import.c]
PyObject* PyImport_GetModuleDict(void)
{
    //1. 通过 PyThreadState_GET()获得当前线程状态对象
    //2. 基于当前线程状态对象获得 PyInterpreterState 对象
    //3. 基于 PyInterpreterState 对象获得其维护的全局 module 集合
    PyInterpreterState *interp = PyThreadState_GET()->interp;
    return interp->modules;
}
```

在 PyImport_AddModule 的代码清单 13-4 的[2]处，会在从[1]处获得的 module 集合中搜索名为 name 的 module 对象，如果该 module 已经存在，则直接将 module 返回；否则，Python 会通过下面的 PyModule_New 创建名为 name 的新的 module 对象，并将（name，module）对应关系插入到 module 集合（interp->modules）中。

```
[moduleobject.c]
typedef struct {
    PyObject_HEAD
    PyObject *md_dict;
} PyModuleObject;

PyObject* PyModule_New(char *name)
{
    PyModuleObject *m;
    PyObject *nameobj;
    m = PyObject_GC_New(PyModuleObject, &PyModule_Type);
    nameobj = PyString_FromString(name);
    m->md_dict = PyDict_New();
    PyDict_SetItemString(m->md_dict, "__name__", nameobj);
    PyDict_SetItemString(m->md_dict, "__doc__", Py_None);
    return (PyObject *)m;
}
```

实际上，PyModuleObject 对象就是对 PyDictObject 对象的一个简单包装，创建 PyModuleObject 对象的动作也显得非常的清晰，注意在这里设置了 module 的__name__属性，却并没有设置其__doc__属性。

我们注意到，PyImport_AddModule 虽然创建了一个 module，但是这仅仅是一个空的 module，并没有包含它所应该包含的操作和数据，而 PyImport_AddModule 似乎也并不在意，在代码清单 13-4 的[3]处将这个空空如也的 PyModuleObject 对象放入到 interp->modules 中就匆匆返回了。

### 13.2.1.2 设置 module 对象

在 PyImport_AddModule 结束后，程序流程回到 Py_InitModule4 手中，在接下来的代码清单 13-4 的[3]处，Py_InitModule4 完成了对__builtin__ module 几乎全部属性的设置。这个属性设置的动作依赖于 Py_InitModule4 的调用者传递进来的第二个参数（methods），在这里为 builtin_methods。Py_InitModule4 会遍历 builtin_methods，并处理其中的每一项元素。我们来看看这个 builtin_methods。

```
[methodobject.h]
typedef PyObject *(*PyCFunction)(PyObject *, PyObject *);

struct PyMethodDef {
    char    *ml_name;      /* The name of the built-in function/method */
    PyCFunction ml_meth;   /* The C function that implements it */
    int     ml_flags;
    char    *ml_doc;    /* The __doc__ attribute, or NULL */
};
typedef struct PyMethodDef PyMethodDef;

[bltinmodule.c]
static PyMethodDef builtin_methods[] = {
    ……
    {"dir",     builtin_dir,       METH_VARARGS, dir_doc},
    ……
    {"getattr", builtin_getattr,    METH_VARARGS, getattr_doc},
    ……
    {"len",     builtin_len,       METH_O, len_doc},
    ……
    {NULL,      NULL},
};
```

看看 builtin_methods 中的家伙啊，dir、getattr、len……个个出身高贵，在 Pythoner 的眼中，它们就代表着 Python。我们从图 13-4 可以形象地看到 PyMethodDef 的结构。

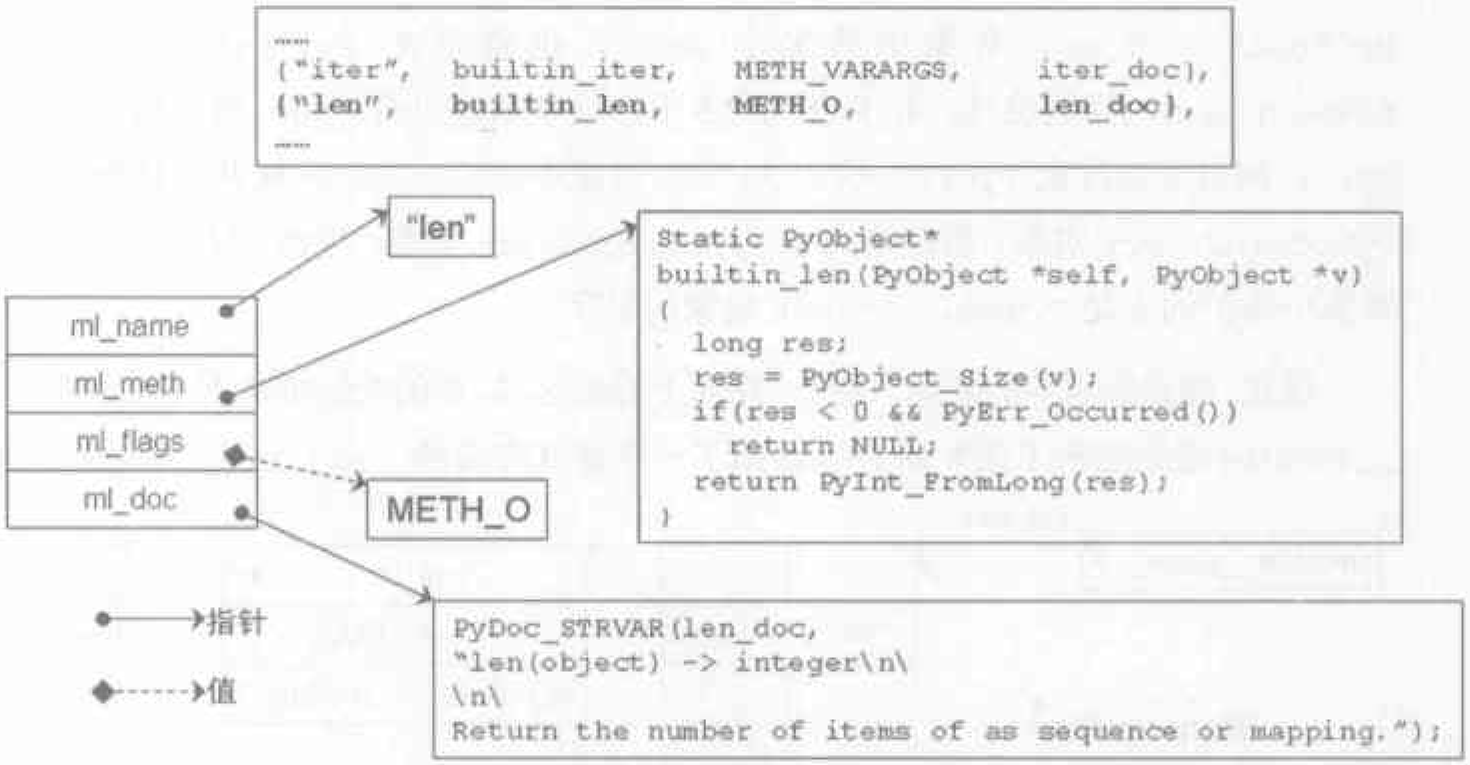



图 13-4 len 函数对应的 PyMethodDef 结构

对于 builtin_methods 中的每一个 PyMethodDef 结构，Py_InitModule4 都会基于它创建一个 PyCFunctionObjet 对象，这个对象是 Python 中对函数指针的包装，当然，还将这个函数指针和其他信息联系在了一起：

```
[methodobject.h]
typedef struct {
    PyObject_HEAD
    PyMethodDef *m_ml; /* Description of the C function to call */
    PyObject    *m_self; /* Passed as 'self' arg to the C func, can be NULL */
    PyObject    *m_module; /* The __module__ attribute, can be anything */
} PyCFunctionObject;

[methodobject.c]
PyObject* PyCFunction_NewEx(PyMethodDef *ml, PyObject *self, PyObject *module)
{
    PyCFunctionObject *op;
    op = free_list;
    if (op != NULL) {
        free_list = (PyCFunctionObject *)(op->m_self);
        PyObject_INIT(op, &PyCFunction_Type);
    }
    else {
        op = PyObject_GC_New(PyCFunctionObject, &PyCFunction_Type);
    }
    op->m_ml = ml;
    op->m_self = self;
    op->m_module = module;
    return (PyObject *)op;
}
```

很显然，Python 对 PyCFunctionObjet 对象采用了缓冲池的策略，不过有了对 PyIntObject，PyListObject 对象的剖析，我们已经可以将这个策略猜个八九不离十了。

PyCFunctionObject 对象中的那个 self，也就是在 Py_InitModule4 中传入的 passthrough，我们说过，在 Python 2.5 中，这个东西没有作用，所以这个 self 通常总为 NULL。同时还要注意 PyCFunctionObject 对象中的 m_module 域并不是指向一个真正的 PyModuleObject 对象，而是指向了 PyStringObject 对象，当然，这个 PyStringObject 对象中维护的正是 PyModuleObject 对象的名字。

现在，翻回到本节的开始，再次一路向下看到这里，你的头脑里是不是已经将__builtin__module 建立起来了呢？图 13-5 给出了一个建立完成的__builtin__module 的示意图。

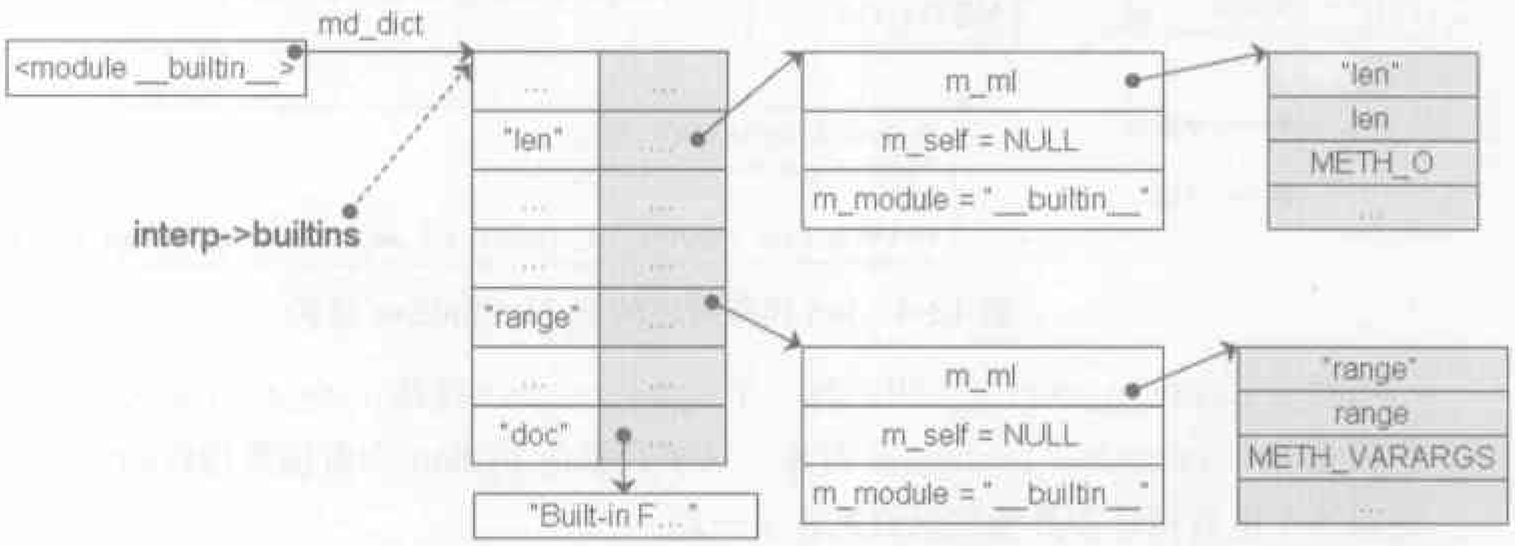



图 13-5 建立完成的__builtin__ module

在_PyBuiltin_Init 之后，Python 将把 PyModuleObject 对象中维护的那个 PyDictObject 对象抽取出来，将其赋给 interp->builtins。这个结果已经在图 13-5 中显示出来了。

```
[moduleobject.c]
PyObject* PyModule_GetDict(PyObject *m)
{
    PyObject *d;
    d = ((PyModuleObject *)m) -> md_dict;
    return d;
}

[pythonrun.c]
void Py_InitializeEx(int install_sigs)
{
    ......
    bimod = _PyBuiltin_Init();
    interp->builtins = PyModule_GetDict(bimod);
    ......
}
```

以后 Python 在需要访问__builtin__module 时，直接访问 interp->builtins 就可以了，不需要再到 interp->modules 中去查找。因为对__builtin__module 的使用在 Python 中会比较频繁，所以这种加速机制是很有效的。

## 13.2.2 创建 sys module

### 13.2.2.1 sys module 的备份

Python 在创建并设置了__builtin__ module 之后，会照猫画虎，用同样的流程设置 sys module，并像设置 interp->builtins 一样设置 interp->sysdict（见代码清单 13-5）。

**代码清单 13-5**

```
[pythonrun.c]
void Py_InitializeEx(int install_sigs)
{
    ......
    //[1]: 创建 sys module
    sysmod = _PySys_Init();
    interp->sysdict = PyModule_GetDict(sysmod);

    //[2]: 备份 sys module
    _PyImport_FixupExtension("sys", "sys");
    ......
}
```

在完成了对__builtin__和 sys 两个 module 的设置之后，PyInterpreterState 对象和 PyThreadState 对象在内存中的情形如图 13-6 所示。

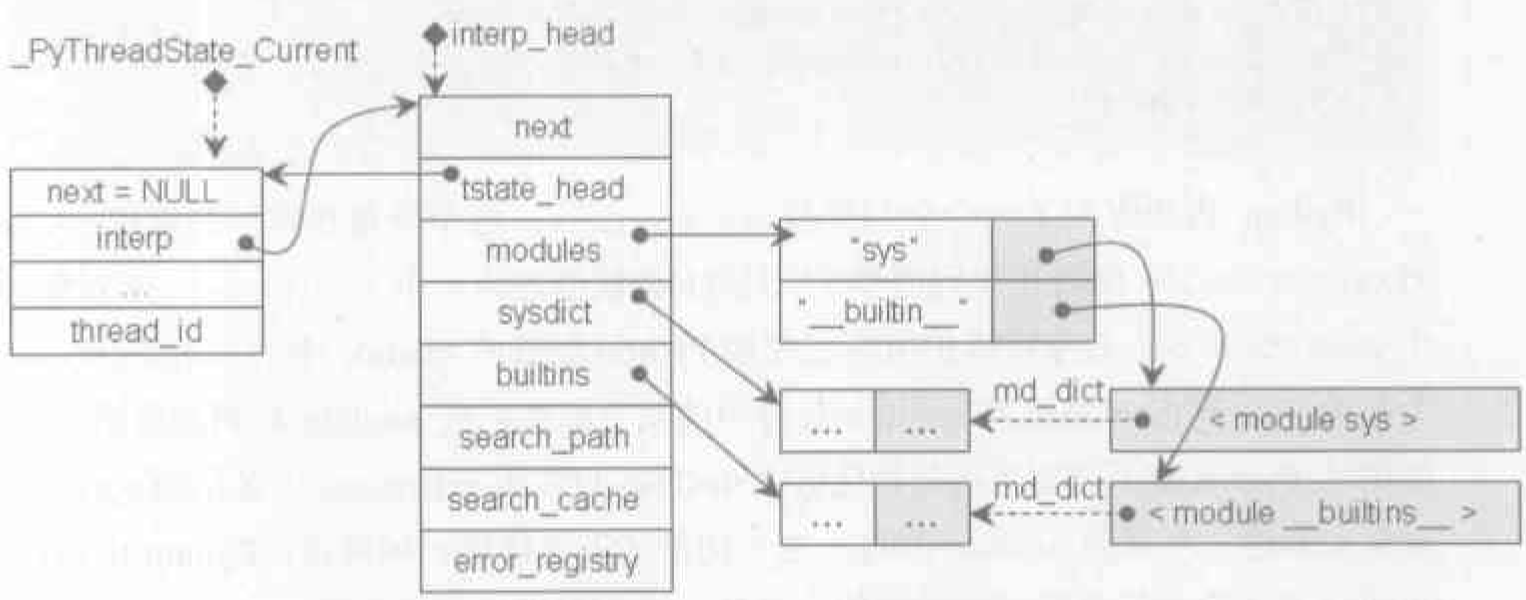



图 13-6 完成 sys module 的创建后的内存布局

注意图 13-6 中从 sys 和__builtin__两个 module 对象中以虚线形式引出的 PyDictObject 对象表示 PyModuleObject 内部维护的那个 PyDictObject 对象（即 md_dict），从前面的分析中我们可以看到，interp->sysdict 和 interp->builtins 指向的确实是 PyDictObject 对象，而非 PyModuleObject 对象。

由于 Python 的 module 集合 interp->modules 是一个 PyDictObject 对象，而 PyDictObject 对象在 Python 中是一个可变对象，所以其中维护的(module name,

PyModuleObject)元素对有可能在运行时被删除。对于 Python 的扩展 module，比如这里的 sys，为了避免对它再一次进行初始化，Python 会将所有的扩展 module 通过一个全局的 PyDictObject 对象来进行备份维护。这个备份的动作在上面的代码清单 13-5 的 Py_InitializeEx 之[2]处通过调用_PyImport_FixupExtension 来完成（见代码清单 13-6）。

**代码清单 13-6**

```
[import.c]
static PyObject *extensions = NULL;

PyObject* _PyImport_FixupExtension(char *name, char *filename)
{
    PyObject *modules, *mod, *dict, *copy;
    //[1]: 如果 extensions 为空，则创建 PyDictObject 对象
    if (extensions == NULL) {
        extensions = PyDict_New();
    }
    //[2]: 获得 interp->modules
    modules = PyImport_GetModuleDict();
    //[3]: 在 interp->modules 中查询以 name 为名的 module
    mod = PyDict_GetItemString(modules, name);
    //[4]: 抽取 module 中的 dict
    dict = PyModule_GetDict(mod);
    //[5]: 对 dict 进行拷贝
    copy = PyDict_Copy(dict);
    //[6]: 将拷贝得到的新的 dict 存储在 extensions 中
    PyDict_SetItemString(extensions, filename, copy);
    return copy;
}
```

Python 内部维护了一个全局变量 extensions，这个变量在第一次调用_PyImport_FixupExtension 时会在代码清单 13-6 的[1]处被创建为一个 PyDictObject 对象，而且这个 PyDictObject 对象将维护所有已经被 Python 加载的 module 中的 PyDictObject 的一个备份。当 Python 系统的 module 集合中的某个标准扩展 module 被删除后不久又被重新加载时，Python 就不需要再次初始化这些 module，只需用 extensions 中备份的 PyDictObject 对象来创建一个新的 module 即可。这一切是基于这样的一种假设：Python 中的标准扩展 module 是不会在运行时动态改变的。显然，这个假设也合情合理。

### 13.2.2.2 设置 module 搜索路径

Python 在创建了 sys module 之后，会在此 module 中设置 Python 搜索一个 module 时的默认搜索路径集合。这个路径集合就是在 Python 执行 import xyz 时将察看的路径的集合：

```
[pythonrun.c]
void Py_InitializeEx(int install_sigs)
{
    ......
```

```
    PySys_SetPath(Py_GetPath());
    ……
}

[sysmodule.c]
void PySys_SetPath(char *path)
{
    PyObject *v;
    v = makepathobject(path, DELIM);
    PySys_SetObject("path", v);
}

int PySys_SetObject(char *name, PyObject *v)
{
    PyThreadState *tstate = PyThreadState_GET();
    PyObject *sd = tstate->interp->sysdict;
    if (v == NULL) {
        if (PyDict_GetItemString(sd, name) == NULL)
            return 0;
        else
            return PyDict_DelItemString(sd, name);
    }
    else
        return PyDict_SetItemString(sd, name, v);
}
```

这里由于篇幅关系，略去了 makepathobject，参考源码，可以看到，在 makepathobject 中，Python 会创建一个 PyListObject 对象，这个 list 中包含了一组 PyStringObject 对象，每一个 PyStringObject 对象的内容就是一个 module 的搜索路径。

最终，这个代表搜索路径集合的 list 对象会在 PySys_SetObject 中被插入到 interp->sysdict 这个 PyDictObject 对象中，在图 13-6 中可以看到，这个 PyDictObject 对象正是在 sys module 中维护的那个 PyDictObject 对象，所以这个搜索路径集合也正是你在 Python 交互式环境下敲入 sys.path 后所看到的那个路径集合。

Python 随后还会进行一些琐碎的动作，其中包括初始化 Python 的 import 环境，初始化 Python 内建异常。初始化 Python 内建异常实际上就是调用 PyType_Ready 初始化各个异常类，而初始化 import 环境会在下一章介绍 import 机制时剖析，这里不再详述。实际上，在考虑 Python 的初始化时，只需有概念上的了解即可。有兴趣的读者可以自己备足粮草，追进代码深处（见代码清单 13-7）。

代码清单 13-7

```
[pythonrun.c]
void Py_InitializeEx(int install_sigs)
{
    //这里设置了 sys.modules，可以看到，它就是 interp->modules
    PyDict_SetItemString(interp->sysdict, "modules", interp->modules);
    //[1]：初始化 import 机制的环境
    _PyImport_Init();
```

```
    //[2]: 初始化 Python 内建 exceptions
    _PyExc_Init();
    //[3]: 备份 exceptions module 和__builtin__ module
    _PyImport_FixupExtension("exceptions", "exceptions");
    /* phase 2 of builtins */
    _PyImport_FixupExtension("__builtin__", "__builtin__");

    //[4]: 在 sys module 中添加一些对象，用于 import 机制
    _PyImportHooks_Init();
    ……
}
```

## 13.2.3 创建__main__ module

在_PyImportHooks_Init()之后，Python 将创建一个非常特殊的 module：一个名为"__main__"的 module（见代码清单 13-8）。

**代码清单 13-8**

```
[pythonrun.c]
static void initmain(void)
{
    PyObject *m, *d;
    //[1]: 创建__main__ module，并将其插入 interp->modules 中
    m = PyImport_AddModule("__main__");
    //[2]: 获得__main__ module 中的 dict
    d = PyModule_GetDict(m);
    if (PyDict_GetItemString(d, "__builtins__") == NULL) {
        //[3]: 获得 interp->modules 中的__builtin__ module
        PyObject *bimod = PyImport_ImportModule("__builtin__");
        //[4]: 将("__builtins__", __builtin__ module)插入到__main__ module 的 dict 中
        PyDict_SetItemString(d, "__builtins__", bimod);
    }
}
```

这个__main__ module 是什么东西呢？我们写一个 Python 程序，有一种最简单的做单元测试的机制，还记得是什么吗？没错，就是那个 if __name__ == "__main__"。前面我们已经看到，在 PyImport_AddModule 时，创建了一个名为 name 的 module 之后，会在该 module 对应的 PyModuleObject 对象的 PyDictObject 对象 md_dict 中插入一个名为"__name__"的项，__main__module 的这一项正是名为"__main__"。作为主程序运行的 Python 源文件就可以被视为名为__main__的 module。

当 Python 以 python abc.py 这样的方式执行时，Python 在沿着名字空间寻找__name__时，就会最终在__main__module 中发现__name__其实为"__main__"；而如果一个 py 文件是以 import 的方式加载的，则__name__不会为"__main__"，也就是说，在 Python 找到__main__module 之前，就已经在某个名字空间找到__name__了。那么奇怪了，无缘无故的，为什么 Python 会找到__main__ module 中去呢？别急，在后面我们介绍初始化

阶段的名字空间时，这个答案就会揭晓。

其实这个__main__ module 我们也是再熟悉不过的了，当进入 Python 交互式环境之后，敲入一个 dir()，输出的结果正是这个__main__ module 的内容，看一看图 13-7，是不是一切都已了然于胸？对于 Python 中那个恼人的问题：__builtin__和__builtins__究竟有什么联系和区别，答案也一目了然了。

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> print __name__
__main__
>>> print __builtins__.__name__
__builtin__
>>>
```

图 13-7 __main__ module 示意图

## 13.2.4 设置 site-specific 的 module 的搜索路径

Python 是一个非常开放的体系，它的强大来源于海量的第三方库，这些库通常由 module 提供，当要使用这些第三方库时，只需简单地进行 import 的动作就 ok 了。一般来说，一些规模较大的第三方库将放在%PythonHome%/lib/site-packages 目录下（%PythonHome%为 Win32 平台上 Python 的安装路径）。若想在程序运行时使用这些库，那么必须使这些库位于 Python 的搜索路径下，Right？

到目前为止，Python 初始化动作只进行了唯一一个与初始化搜索路径集合相关的动作：PySys_SetPath(Py_GetPath())。但是不幸的是，这个设置动作并没有将 site-packages 包含在内。在完成了__main__ module 的创建之后，Python 才腾出手来，收拾这个 site-packages。这个动作的关键在于%PythonHome%/lib 目录下的一个标准库：site.py。

我们先来做个实验，将 site.py 内的内容全部替换为以下内容：

```
print 'we are in site.py'
```

在我的 Python 中，安装了 Django。在图 13-8 中，显示了改动前后 import django 的运行情况。

```
Python 2.5(r25…
>>> import django
>>> print django.VERSION
(0, 96, None)
```

改动site.py前

```
we are in site.py
Python 2.5 (r25…
>>> import django
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named django
```

改动site.py后

图 13-8 改动 site.py 前后对 django 的加载情况

可以看到，Python 在初始化过程中确实进入了 site.py，所以才有了右侧的输出。而这个 site.py 也正是 Python 能正确加载位于 site-packages 目录下的 django 的关键。我们可以猜测，应该就是这个 site.py 将 site-packages 目录加入到了前面提到的 sys.path 中。而这个动作是由 initsite 完成的。

```
[pythonrun.c]
void Py_InitializeEx(int install_sigs)
{
    ……
    _PyImportHooks_Init();
    initmain(); /* Module __main__ */
    initsite(); /* Module site */
    ……
}
```

```
[pythonrun.c]
static void initsite(void)
{
    PyObject *m;
    m = PyImport_ImportModule("site");
}
```

在 initsite 中，只调用了 PyImport_ImportModule 函数，这个函数是 Python 中 import 机制的核心所在，将在下一章详细剖析。在这里，我们只需要知道，这个函数调用，可以简化为一行 Python 代码："import site"。就是在这里，Python 进入了 site.py，从而也就能输出图 13-8 右侧的"we are in site.py"。

在 site.py 中，Python 进行了两个动作。

1. 将 site-pakcages 路径加入到 sys.path 中，对于不同平台，又分不同情况。
    - Win32 平台：%PythonHome%/lib/site-packages。
    - Unix/Linux 平台：
        - %sys.prefix%/lib/python<version>/site-packages（其中%sys.prefix%为 Python 的 sys.prefix）；
        - %sys.prefix%/lib/site-python；
        - %sys.exec_prefix%/lib/python<version>/site-packages；
        - %sys.exec_prefix%/lib/site-python。
2. 处理 site-packages 目录下的所有.pth 文件中的所有路径加入到 sys.path 中。

对于第 2 条，来个具体的例子，在我的 site-packages 目录下，有一个 wx.pth，是安装 wxPython 时装入的路径文件。wx.pth 的内容如下：

```
[wx.pth]
wx-2.8-msw-unicode
```

所以，我的 sys.path 的结果如下：

```
>>> import sys
>>> sys.path
['', 'C:\\WINDOWS\\system32\\python25.zip', 'c:\\python25\\DLLs',
 'c:\\python25\
\lib', 'c:\\python25\\lib\\plat-win', 'c:\\python25\\lib\\lib-tk',
 'c:\\python25
', 'c:\\python25\\lib\\site-packages',
 'c:\\python25\\lib\\site-packages\\wx-2.8
-msw-unicode']
>>>
```

好了，现在你可以自己在 site-packages 中写一个 module，然后添加一个 pth 文件，看看能否正确地在 Python 中对你的 module 进行 `import` 动作。

到现在，Python 中绝大部分重要的初始化动作都已经完成了，好了，我们来看一看完成这些初始化动作之后 Python 为我们准备了什么重要的东西，这些东西就是我们在正式运行 Python 程序时可以利用的资源，如图 13-9 所示。

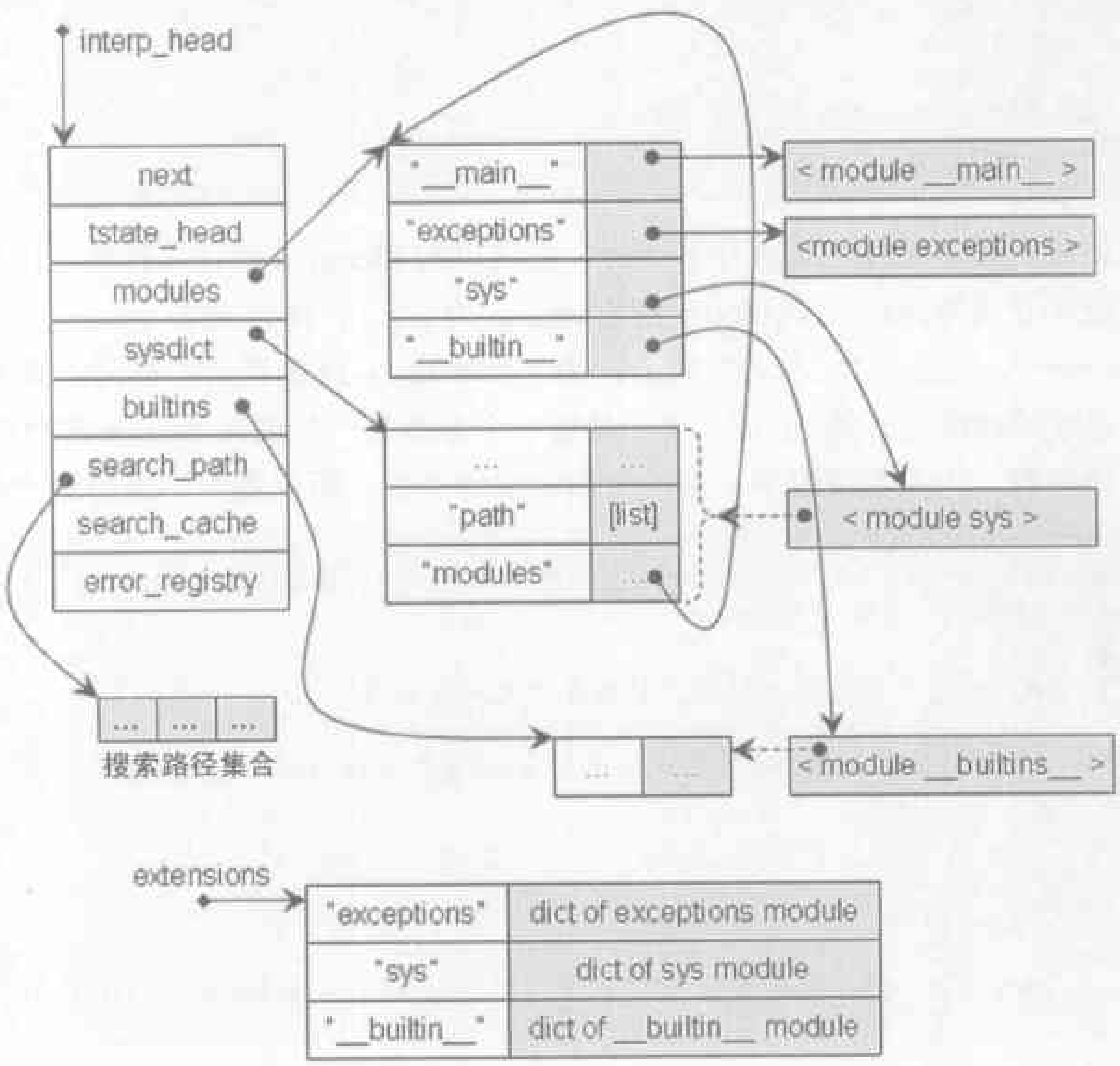



图 13-9 完成初始化后的环境

## 13.3 激活 Python 虚拟机

到了这里，Python 已经完成了执行程序所必需的基础设施的建设，但是我们认为，初始化的动作还没有真正完成，当 Python 真正地进入到了我们在第 2 部分中详细剖析的字节码虚拟机后，初始化的阶段才算真正完成。

一般地，Python 有两种运行的方式，一种是在命令行下的交互式环境；另一种则是以 python abc.py 的方式运行脚本文件。看上去这两种方式有些不同，但实际上，正如我们马上将要看到的，这两种执行方式将殊途同归，进入同一个字节码虚拟机。

Python 在 Py_Initialize 成功完成之后，最终将调用 PyRun_AnyFileExFlags：

```
[main.c]
int Py_Main(int argc, char **argv)
{
    Py_Initialize();
    ......
    PyRun_AnyFileExFlags(
        fp,
        filename == NULL ? "<stdin>" : filename,
        filename != NULL, &cf);
    ......
}
```

如果以脚本文件方式运行 Python，那这里的 filename 就是文件名，比如 abc.py；而如果是以交互式方式运行 Python，则 filename 为 NULL，所以会为 PyRun_AnyFileExFlags 传入一个“<stdin>”。与之对应的，第一个参数 fp 或是指向了打开的脚本文件，或是指向了系统的标准输入流 stdin。至于最后一个参数是一些编译 Python 源代码时可能会用到的编译参数，大多数情况下，都不会指定编译参数，所以我们不考察这个参数。

```
[pythonrun.c]
int PyRun_AnyFileExFlags(FILE *fp, const char *filename, int closeit,
 PyCompilerFlags *flags)
{
    //根据 fp 是否代表交互环境，对程序流程进行分流
    if (Py_FdIsInteractive(fp, filename)) {
        int err = PyRun_InteractiveLoopFlags(fp, filename, flags);
        if (closeit)
            fclose(fp);
        return err;
    }
    else
        return PyRun_SimpleFileExFlags(fp, filename, closeit, flags);
}
```

Python 通过 Py_FdIsInteractive 来判断 fp 是否指向了标准输入流，如果是，则表明 Python 是以交互式的方式运行的，从而进入 PyRun_InteractiveLoopFlags；而脚本文件则进入另一条路径 PyRun_SimpleFileExFlags，别着急，过一会就可以看到，两条路径最终又会融入同一条路径。

## 13.3.1 交互式运行方式

我们先来看看 Python 以交互式方式运行时的情形（见代码清单 13-9）。

**代码清单 13-9**

```
[pythonrun.c]
int PyRun_InteractiveLoopFlags(FILE *fp, const char *filename,
PyCompilerFlags *flags)
{
    PyObject *v;
    int ret;
    //[1]: 创建交互式环境提示符 ">>> "
    v = PySys_GetObject("ps1");
    if (v == NULL) {
        PySys_SetObject("ps1", v = PyString_FromString(">>> "));
    }
    //[2]: 创建交互式环境提示符 "... "
    v = PySys_GetObject("ps2");
    if (v == NULL) {
        PySys_SetObject("ps2", v = PyString_FromString("... "));
    }
    //[3]: 进入交互式环境
    for (;;) {
        ret = PyRun_InteractiveOneFlags(fp, filename, flags);
        if (ret == E_EOF)
            return 0;
    }
}

int PyRun_InteractiveOneFlags(FILE *fp, char *filename, PyCompilerFlags
 *flags)
{
    PyObject *m, *d, *v, *w;
    mod_ty mod;
    PyArena *arena;
    char *ps1 = "", *ps2 = "";

    v = PySys_GetObject("ps1");
    if (v != NULL) {
        ps1 = PyString_AsString(v);
    }
    w = PySys_GetObject("ps2");
    if (w != NULL) {
        ps2 = PyString_AsString(w);
    }
    //[4]: 编译用户在交互式环境下输入的 Python 语句
    arena = PyArena_New();
    mod = PyParser_ASTFromFile(fp, filename,
                Py_single_input, ps1, ps2,
                flags, &errcode, arena);

    //获得<module __main__>中维护的 dict
    m = PyImport_AddModule("__main__");
```

```
    d = PyModule_GetDict(m);
    //[5]: 执行用户输入的 Python 语句
    v = run_mod(mod, filename, d, d, flags, arena);
    PyArena_Free(arena);
    return 0;
}
```

在 PyRun_InteractiveLoopFlags 中，可以发现 Python 在代码清单 13-9 中的[1]和[2]处分别设置了在交互式环境中每天和我们“face to face”的两个提示符。然后，在[3]处调用的 PyRun_InteractiveOneFlags 中，Python 完成了至关重要的两步，如代码清单 13-9 所示。

代码清单 13-9 的[4]和[5]分别有如下含义：

[4] 调用 PyParser_ParseFileFlags，对用户在交互式环境下输入的 Python 语句进行编译，其结果是构造与 Python 语句对应的抽象语法树（AST），并返回 AST。

[5] 调用 run_mode，在 run_mode 中，将最终完成对用户输入语句的执行动作。关于这一点，后面我们会看得很清楚。

注意到 Python 在进入 run_mode 之前，会将__main__ module 中维护的 PyDictObject 对象取出，作为参数传递给 run_mode，这个参数关系极为重大，实际上这里的参数 d 就将作为 Python 虚拟机开始执行时当前活动的 frame 对象的 local 名字空间和 global 名字空间。

## 13.3.2 脚本文件运行方式

接下来，我们看一看直接运行脚本文件的方式（见代码清单 13-10）。

**代码清单 13-10**

```
[python.h]
#define Py_file_input 257

[pythonrun.c]
int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit,
        PyCompilerFlags *flags)
{
    PyObject *m, *d, *v;
    const char *ext;
    //[1]: 在__main__ module 中设置 "__file__" 属性
    m = PyImport_AddModule("__main__");
    d = PyModule_GetDict(m);
    if (PyDict_GetItemString(d, "__file__") == NULL) {
        PyObject *f = PyString_FromString(filename);
        PyDict_SetItemString(d, "__file__", f);
    }
    //[2]: 执行脚本文件
```

```
    v = PyRun_FileExFlags(fp, filename, Py_file_input, d, d, closeit, flags);
    ......
}

PyObject *
PyRun_FileExFlags(FILE *fp, const char *filename, int start, PyObject
*globals,PyObject *locals, int closeit, PyCompilerFlags *flags)
{
    PyObject *ret;
    mod_ty mod;
    PyArena *arena = PyArena_New();
    //编译
    mod = PyParser_ASTFromFile(fp, filename, start, 0, 0, flags, NULL, arena);
    if (closeit)
        fclose(fp);
    //执行
    ret = run_mod(mod, filename, globals, locals, flags, arena);
    PyArena_Free(arena);
    return ret;
}
```

很显然，脚本文件的执行流程虽然和交互式执行方式的流程在 PyRun_SimpleFileExFlags 中被分流了，但是，它们都有着相同的动作。同交互式执行方式一样，脚本文件的执行流程最后也进入了 run_mode，而且也同样地将__main__module 中维护的 PyDictObject 对象作为 local 名字空间和 global 名字空间传入了 run_mode。在这里，我们可以更清楚地通过调用 run_node 时的变量名看到这一点。看看那个 run_mode 中的参数 globals、locals，再明显不过了。

好了，欢呼吧，在 PyRun_AnyFileExFlags 中，两种执行方式分道扬镳，到了这里，又胜利会师，现在是一起杀入 run_mode 的时候了。

### 13.3.3 启动虚拟机

从 run_mode 开始，Python 现在只剩下最后一件需要完成的工作了，那就是启动字节码虚拟机，开始让 Python 成为一个有生命的精灵（见代码清单 13-11）。

**代码清单 13-11**

```
[pythonrun.c]
static PyObject *
run_mod(mod_ty mod, const char *filename, PyObject *globals, PyObject
  *locals, PyCompilerFlags *flags, PyArena *arena)
{
    PyCodeObject *co;
    PyObject *v;
    //[1]: 基于 AST 编译字节码指令序列，创建 PyCodeObject 对象
    co = PyAST_Compile(mod, filename, flags, arena);
    //[2]: 创建 PyFrameObject 对象，执行 PyCodeObject 对象中的字节码指令序列
    v = PyEval_EvalCode(co, globals, locals);
```

```
    Py_DECREF(co);
    return v;
}
```

在代码清单 13-11 的[1]处，run_mode 接过传入的 AST，倒手就传入 PyAST_Compile 中，在这个函数中，Python 基于 AST，最终完成了字节码的编译工作，并且创建了一个我们已经非常熟悉的 PyCodeObject 对象。至于完整的编译过程，这又是另一个重大的话题了。

而在接下来的代码清单 13-11 的[2]处，Python 已经做好了一切准备工作，开始通过 PyEval_EvalCode 着手唤醒字节码虚拟机。

```
[ceval.c]
PyObject* PyEval_EvalCode(PyCodeObject *co, PyObject *globals, PyObject
  *locals)
{
       return PyEval_EvalCodeEx(co,
             globals, locals,
             (PyObject **)NULL, 0,
             (PyObject **)NULL, 0,
             (PyObject **)NULL, 0,
             NULL);
}

PyObject *
PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
      PyObject **args, int argcount, PyObject **kws, int kwcount,
      PyObject **defs, int defcount, PyObject *closure)
{
    register PyFrameObject *f;
    register PyObject *retval = NULL;
    register PyObject **fastlocals, **freevars;
    PyThreadState *tstate = PyThreadState_GET();
    PyObject *x, *u;
    ......
    f = PyFrame_New(tstate, co, globals, locals);
    ......
    fastlocals = f->f_localsplus;
    ......
    retval = PyEval_EvalFrameEx(f, 0);
    return retval;
}
```

从操作系统为 Python 创建进程开始，到这里，经过了如此多的跌跌撞撞，我们终于看到了黎明的曙光。在这里，我们看到了在分析函数机制时和我们朝夕相处的 PyEval_EvalCodeEx，看到了已经深深印入我们脑海的 PyFrameObject 对象，看到了那个掌控 Python 世界中无数对象生生灭灭的字节码虚拟机——PyEval_EvalFrameEx。我们曾经在那里探索了很久，挖掘了很久，现在，我们再一次回到起始的地方，终于有了一种融会贯通的顿悟。

从 Python 进程被创建，到 Python 字节码虚拟机被唤醒，再到之后执行引擎循环往复地执行字节码，这个过程已经清晰地展现出来了。虽然还有很多细节隐藏在幕后，但是对于 Python 的骨架，我们已经看清了，到了此时，Python 再也不是什么神秘的东西了，它应该从云端落到地面，成为你手中的玩具了☺。

## 13.3.4 名字空间

好了，现在我们来玩点有趣的东西，来看一看激活字节码虚拟机的过程中，在创建 PyFrameObject 对象时，所设置的 3 个名字空间：local、global、builtin（见代码清单 13-12）。

**代码清单 13-12**

```
[frameobject.c]
PyFrameObject *
PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject *globals,
        PyObject *locals)
{
    PyFrameObject *back = tstate->frame;
    PyFrameObject *f;
    PyObject *builtins;
    int extras, ncells, nfrees, i;

    //[1]: 设置 builtin 名字空间
    if (back == NULL || back->f_globals != globals) {
        builtins = PyDict_GetItem(globals, builtin_object);
    }
    else {
        builtins = back->f_builtins;
    }
    f->f_builtins = builtins;
    f->f_back = back;

    //[2]: 设置 global 名字空间
    f->f_globals = globals;

    //[3]: 设置 local 名字空间
    if ((code->co_flags & (CO_NEWLOCALS | CO_OPTIMIZED)) ==
        (CO_NEWLOCALS | CO_OPTIMIZED))
        locals = NULL; //调用函数，不需创建 local 名字空间
    else if (code->co_flags & CO_NEWLOCALS) {
        locals = PyDict_New();
    }
    else {
        if (locals == NULL)
            locals = globals; //一般情况下，locals 和 globals 指向形同的 dict
    }
        f->f_locals = locals;
    ......
```

```
    return f;
}
```

这里的 tstate 是从 PyEval_EvalCodeEx 中传入的，实际上也就是在图 13-3 中所示的那个_PyThreadState_Current：

```
[ceval.c]
PyObject *
PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
        PyObject **args, int argcount, PyObject **kws, int kwcount,
        PyObject **defs, int defcount, PyObject *closure)
{
    ......
    PyThreadState *tstate = PyThreadState_GET();
    ......
    f = PyFrame_New(tstate, co, globals, locals);
    ......
}
```

显然，在 PyFrame_New 的代码清单 13-12 的[1]处设置 builtin 名字空间时，这里的 back(tstate->frame)是 NULL，因为这是 Python 为当前线程创建的第一个 PyFrameObject 对象，所以 builtins = PyDict_GetItem(globals, builtin_object)一定会执行。在前面我们已经知道，这个 globals 就是__main__ module 中维护的 dict，那 builtin_object 是什么东西呢？还记得吗，在分析 Py_InitializeEx 时，我们曾经提到过它，现在我们来复习一下：

```
[frameobject.c]
static PyObject *builtin_object;

int _PyFrame_Init()
{
    builtin_object = PyString_InternFromString("__builtins__");
    return (builtin_object != NULL);
}
```

显然，builtin_object 是 PyStringObject 对象，而其维护的字符串是“__builtins__”，再看看图 13-7 中，通过执行 dir() 显示出来的__main__module 中 dict 的内容，没错，正是有一个“__builtins__”，所以在 PyFrame_New 中设置的 builtin 名字空间实际上就是我们在前面已经照过很多次面的__builtin__ module。

当然，如果 back 并不为空，那么 builtins 将是 back->builtins，稍一推理，我们就能发现，这种机制意味着这样一个事实：**Python 所有的线程都共享同样的 builtin 名字空间。**

同样，我们可以看到在 PyFrame_New 中的代码清单 13-12 的[2]处，globals 被设置成了__main__module 中的 dict，而[3]处对 local 名字空间的设置则复杂了很多，但是在这里（激活 Python 字节码虚拟机时），local 名字空间和 global 名字空间一样，也被设置成了__main__module 中的 dict。

在 Python 源代码中添加输出信息，当执行任意一个 Python 脚本文件 demo.py 时，可以看到输出结果正如我们所预期的那样，如图 13-10 所示。

```
+++++++++++++++++++++++ f->f_builtins ++++++++++++++++++++++
... input ... round dir range ... AttributeError OverflowError WindowsError

+++++++++++++++++++++++ f->f_globals ++++++++++++++++++++++
__builtins__ __name__ __file__ __doc__

+++++++++++++++++++++++ f->f_locals ++++++++++++++++++++++
__builtins__ __name__ __file__ __doc__
```

图 13-10 查看 Python 虚拟机启动时的名字空间

由于__builtin__module 中的属性太多，所以这里只显示了一部分。有一点奇怪的是，我们在前面看到，在 Py_Initialize 完成时，__main__ module 中似乎并没有__file__属性。没错，实际上，这个属性是在 PyRun_SimpleFileExFlags 中加入的，可以参考前面关于 PyRun_SimpleFileExFlags 的代码。与图 13-10 对应，图 13-11 显示了交互式环境下的 local 名字空间和 global 名字空间。

```
>>> showDict(globals())
('showDict', <function showDict at 0x00CD7630>)
('__builtins__', <module '__builtin__' (built-in)>)
('__name__', '__main__')
('__doc__', None)
>>> showDict(locals())
('showDict', <function showDict at 0x00CD7630>)
('__builtins__', <module '__builtin__' (built-in)>)
('__name__', '__main__')
('__doc__', None)
```

图 13-11 交互式环境下的 local 名字空间和 global 名字空间

现在我们可以从理论上推导一下，当执行 dir()时，Python 执行引擎首先要找到符号“dir”对应的对象。从图 13-10 可以看到，这个符号位于 builtin 名字空间中。而且，实际上它对应着 bltmodule.c 中的 builtin_dir 函数，这个函数将显示 local 名字空间中的属性。我们知道，名字空间这个概念在 Python 中实际就是一个 dict，所以 dir()实际显示的就是对应 local 名字空间的 dict 的键的集合。同样是在图 13-10 中，可以看到，local 名字空间实际就是__main__module 中的那个 dict，所以 dir 的输出你是不是已经猜到了呢？显然，在交互式环境下输入 dir 的话，这个执行的过程还是一样的，但是正如图 13-11 所示的情况，local 名字空间中不会再有__file__这个属性了，还记得吗，原因就在于交互式方式曾经有一段时间和脚本文件执行方式分道扬镳。

CHAPTER

第 14 章

# Python 模块的动态加载机制

在之前的章节中，我们考察的东西都是局限在一个模块（在 Python 中，就是 module）内。然而一个现实中的程序不可能只有一个模块，更多的情况下，一个程序会有多个模块，而模块之间存在着引用和交互，这些引用和交互也是程序的一个重要组成部分。本章剖析的就是在 Python 中，一个模块是如何加载、引用另一个模块中的功能的。我们的研究从 Python 中的一个 module 如何从硬盘中被加载到内存中开始。

## 14.1 import 前奏曲

我们从 import.py 开始研究 Python 中对 module 的动态加载机制(以后简称动态加载机制)，对 Python 中动态加载机制的深入理解也是编写 Python 扩展模块的关键所在，当你明了 Python 是如何基于硬盘上的 py 文件或 dll（注意：在 Python 2.5 中，对 dll 后缀名的支持被删除了，但是可以将 dll 后缀改为 pyd 后缀，以重新获得 Python 2.5 的支持。尽管如此，在后续的描述中，我们依然称 dll 文件而非 pyd 文件）文件中的内容来创建 Python 可识别的运行时模块后，编写 Python 的扩展模块自然就是水到渠成的事了。

```
[import.py]
import sys
0  LOAD_CONST      0 (-1)
3  LOAD_CONST      1 (None)
6  IMPORT_NAME     0 (sys)
9  STORE_NAME      0 (sys)
```

这个 import.py 文件简单无比，所以其编译之后所产生的常量表（co_consts）和符号表（co_names）也是极其地简单，如图 14-1 所示。

```
- <consts>
    <int value="-1" />
    <NoneObject />
  </consts>
- <names>
    <internStr index="0" length="3" value="sys" />
  </names>
```

图 14-1 import.py 编译结果中的常量表和符号表

可以看到，import 的结果最终将导致 Python 虚拟机通过指令“9 STORE_NAME 0”将 sys module 存储在当前 PyFrameObject 的 local 名字空间中。当在 import 之后使用 sys module，比如执行“print sys.path”时，Python 虚拟机就能很轻松地找到“sys”这个符号了。

LOAD_CONST 指令和 STORE_NAME 指令都是我们非常熟悉的了，显然这些通用指令与 Python 的 import 机制并没有什么太大的联系。其实从指令的名字就能看出，IMPORT_NAME 指令一定是与 Python 的 import 机制息息相关的。我们来看看 Python 对于字节码 IMPORT_NAME 的实现（见代码清单 14-1）。

**代码清单 14-1**

```
[IMPORT_NAME]
        w = GETITEM(names, oparg);
        x = PyDict_GetItemString(f->f_builtins, "__import__");
        v = POP();
        u = TOP();
        //[1]: 将 Python 的 import 动作需要使用的信息打包到 tuple 中
        if (PyInt_AsLong(u) != -1 || PyErr_Occurred()) {
            w = PyTuple_Pack(5,
                    w,
                    f->f_globals,
                    f->f_locals == NULL ? Py_None : f->f_locals,
                    v,
                    u);
        } else {
            w = PyTuple_Pack(4,
                    w,
                    f->f_globals,
                    f->f_locals == NULL ? Py_None : f->f_locals,
                    v);
        }
        x = PyEval_CallObject(x, w);
        SET_TOP(x);
```

在代码清单 14-1 的[1]之前，w 是 PyStringObject 对象“sys”，v 是通过“3 LOAD_CONST 1”指令被压入到运行时栈中的 PyNone，而 u 则是“0 LOAD_CONST 0”指令被压入到运行时栈的那个-1，为什么莫名其妙钻出来个-1，在以后的剖析中我们会知道原因。现在，w、v、u 的身份都查明白了，那么那个 x 又是什么呢？

在上一章对 Python 初始化的分析中，我们看到，f->f_builtins 实际上就是

__builtin__module 所维护的那个 dict，而其中的“__import__”符号，正对应 bltinmodule.c 中的 builtin___import__函数。不过在上一章分析 Python 的初始化动作时，我们已经看到，在初始化__builtin__module 时，这个函数已经摇身一变，被包装成了一个 PyCFunction-Object 对象了。所以这里的 x 对应的就是这个 PyCFunctionObject 对象。

在代码清单 14-1 的[1]处，有一个依据 u 的值而进行的判断，在我们的例子中，u 为-1，所以程序流程进入的是 else 下的分支，另一分支以后会详细介绍。Python 将 w、v 和当前活动的 PyFrameObject 对象中的 global 名字空间、local 名字空间一起打包，做成了一个 PyTupleObject 对象，其中包含了 Python 在此后进行 import 动作时所需的所有信息。在创建了这个 tuple 对象之后，它和代表 import 操作的 PyCFunctionObjet 对象一起联手打入了 PyEval_CallObject：

```
[cevcl.c]
#define PyEval_CallObject(func,arg) \
      PyEval_CallObjectWithKeywords(func, arg, (PyObject *)NULL)

PyObject* PyEval_CallObjectWithKeywords(PyObject *func, PyObject *arg,
PyObject *kw)
{
    PyObject *result;

    if (arg == NULL)
        arg = PyTuple_New(0);
    else if (!PyTuple_Check(arg)) {
        PyErr_SetString(PyExc_TypeError, "argument list must be a tuple");
        return NULL;
    }

    if (kw != NULL && !PyDict_Check(kw)) {
        PyErr_SetString(PyExc_TypeError, "keyword list must be a
          dictionary");
        return NULL;
    }

    result = PyObject_Call(func, arg, kw);
    return result;
}
```

这里的 arg 就是前面刚刚打包好的 PyTupleObject 对象，PyEval_CallObjectWith-Keywords 仅仅是简单地检查了参数的有效性，然后就调用了 PyObject_Call。

这个 PyObject_Call 我们可是非常熟悉了啊，以前在剖析 Python 函数机制时就和这位老兄打了不少交道。我们知道，PyObject_Call 是一个相当范型的函数，它将对一切可调用的（callable）对象进行“调用”操作。具体地说，最终 PyObject_Call 将调用 func 参数对应的类型对象中所定义的 tp_call 操作。

我们刚才提到，这个 func 对象实际上是一个 PyCFunctionObject 对象，那么它对应的类型对象是什么呢？正是 PyCFunction_Type：

```
[methodobject.c]
PyTypeObject PyCFunction_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "builtin_function_or_method",
    ……
    PyCFunction_Call,           /* tp_call */
    ……
};
```

在 PyCFunction_Type 中，tp_call 被设置成了 PyCFunction_Call。如此说来，对于 Python 虚拟机而言，这个 PyCFunctionObject 对象确实是一个可调用的对象，好，那就沿着它给我们指示的 PyCFunction_Call 函数，一路追踪下去（见代码清单 14-2）。

**代码清单 14-2**

```
[methodobject.c]
PyObject* PyCFunction_Call(PyObject *func, PyObject *arg, PyObject *kw)
{
    PyCFunctionObject* f = (PyCFunctionObject*)func;
    PyCFunction meth = PyCFunction_GET_FUNCTION(func);
    PyObject *self = PyCFunction_GET_SELF(func);
    int size;

    switch (PyCFunction_GET_FLAGS(func) & ~(METH_CLASS | METH_STATIC |
    METH_COEXIST)) {
    case METH_VARARGS:
        if (kw == NULL || PyDict_Size(kw) == 0)
            return (*meth)(self, arg);
        break;
    case METH_VARARGS | METH_KEYWORDS:
    case METH_OLDARGS | METH_KEYWORDS:
        //[1]: 函数调用
        return (*(PyCFunctionWithKeywords)meth)(self, arg, kw);
    ……
    }
    PyErr_Format(PyExc_TypeError, "%.200s() takes no keyword arguments",
            f->m_ml->ml_name);
    return NULL;
}
```

在 PyCFunction_Call 中，Python 虚拟机从 PyCFunctionObject 对象中抽取出了它所维护的那个函数指针——meth，这个指针也就是指向 C 函数 builtin___import__的函数指针。然后，Python 虚拟机直接利用之前创建的那个 PyTupleObject 对象 arg 调用这个函数指针所指向的函数——builtin___import__。到了这里，真正实现 import 机制的操作手被我们找到了，Python 从接收到 import 的指令开始，到现在，才真正地找准目标，备足粮草，开始真枪实弹地进行 import 动作了。

我们知道，在 Python 中，从语法层面讲，`import` 有很多种写法，比如：

```
import sys
import xml.sax
from sys import path
from sys import path as mypath
import usermodule
```

从 `import` 的目标来说，可以有系统的 standard module，还有用户自己写的 module，而用户 module 中又分 Python 语言写的 module 和 C 语言写的以 dll 形式存在的 module。我们将一一介绍各种语法，各种目标 module 背景下 Python 的 `import` 动作。考虑到尽可能简洁，在展示代码时，我们可能会对代码进行一些删节，比如一个函数中可能会处理 `import sys` 这样的语法，也会处理 `import xml.sax` 这样的语法，那我们在展示代码时，如果考虑的只是 `import sys` 这样的语法，则会删节处理 `import xml.sax` 这样语法的代码。为了更好地理解 Python 的 import 机制，请在阅读时对照参考 Python 源代码。

## 14.2 Python 中 import 机制的黑盒探测

同 Java 中的 package 机制、C++中的 namespace 机制一样，Python 通过 module 机制和稍后会挖掘的 package 机制来实现对系统复杂度的分解，以及保护名字空间不受污染。

通过 module 和 package，我们可以将某个功能、某种抽象进行独立的实现和维护，在 module 和 package 的基础之上构建软件，这样不仅使得软件的架构清晰，而且也能很好地实现代码复用。

在 Python 中，module 和 package 通过 import 机制融入 Python，本节中，我们将对 Python 中的 import 机制进行黑盒探测。所谓黑盒探测，指的是我们将通过各种不同的 import 动作观察 Python 在动态加载 module 或 package 时会进行什么样的动作，以及将对执行环境产生什么样的影响。

在本节中，我们并不涉及 Python 中的 import 机制是如何实现的，而是通过这样的黑盒，对 import 机制建立一个整体的框架性的认识，为下一节开始的对 import 机制的白盒探测——也就是源码分析——打下坚实的基础。

### 14.2.1 标准 import

#### 14.2.1.1 Python 内建 Module

对于 Python 用户，`sys module` 恐怕是一个非常熟悉也使用得非常频繁的 module，我们对 import 机制的黑盒探测也从 `sys module` 开始。

在对 Python 运行环境的初始化的分析中，我们看到了 dir，这个小工具是我们探测 import 机制的终极法宝。如果你在 Python 的交互式环境下键入 help(dir)，你会看到在执行 dir 操作时，如果没有参数，则会打印出当前的 local 名字空间的所有符号，而如果有参数，则会将参数视为对象，输出该对象的所有属性。我们的第一个探测就是看一看 import 动作对当前名字空间的影响，如图 14-2 所示。

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> import sys
>>> dir()
['__builtins__', '__doc__', '__name__', 'sys']
>>> type(sys)
<type 'module'>
```

图 14-2 引入 sys module 对名字空间的影响

当 Python 初始化完成之后，当前名字空间中只有“__builtins__”、“__doc__”、“__name__”3 个符号，而在进行了 import 的动作之后，当前 local 名字空间中增加了一个“sys”的符号，而且通过 type 操作，我们发现，在这个“sys”的符号背后，实际上隐藏着一个 module 对象，在 Python 内部，实际上就是一个 PyModuleObject 对象。很显然，Python 中的 import 机制影响了当前 local 名字空间，使得加载的 module 在 local 名字空间成为可见的，而引用该 module 的方法正式通过 module 的名字，即这里的“sys”。

细心的读者一定还记得在之前对 Python 运行环境初始化的分析中，我们发现 Python 在初始化时，就将 sys module 加载到了内存中，实际上，Python 是将一大批的 module 加载到了内存中。但是为了使 local 名字空间能够达到最干净的效果，Python 并没有将这些符号暴露在当前的 local 名字空间中，而是需要用户显式地通过 import 机制通知 Python：我需要将这个符号引入到 local 名字空间中，以便我的程序使用这个符号背后的对象。

这些预先被加载进内存的 module 存放在 sys.modules 中，下面的图 14-3 展示了我们如何将存放在 sys.modules 中的 os 引入到 local 名字空间中。

```
>>> def show_modules():
        for item in sys.modules.items():
            print item

>>> show_modules()
......
('random', <module 'random' from 'C:\Python25\lib\random.pyc'>)
......
('os', <module 'os' from 'C:\Python25\lib\os.pyc'>)
......
>>> id(sys.modules['os'])
11474832
>>> import os
>>> id(os)
11474832
>>> dir()
['__builtins__', '__doc__', '__name__', 'os', 'show_modules', 'sys']
```

图 14-3 探索对 os module 的加载

可以看到，os module 是从 C:\Python25\lib\os.pyc 这个文件中引入的，它是一个 Python 的内建 module。通过 import 机制，这个 module 被引入到了当前的 local 名字空间中。从两个不同的 id 操作中可以看到，毫无疑问，Python 虚拟机的 import 动作引入的正是事先已经在 Python 初始化阶段被加载到 sys.modules 集合中的 os module。

### 14.2.1.2 用户自定义 Module

接下来，我们看一看当 Python 对非内建的 module 进行 import 时所发生的动作，在 Python 中，用户可以通过 py 文件创建自己的 module，也可以通过 C 语言创建 dll，生成 Python 的扩展 module，这些都不是 Python 的内建 module。因为我们在这一节只是做 import 机制的框架性黑盒探测，并不探讨 import 机制的实现，所以这里我们并不区分 py 文件和 dll 文件，只以 py 文件作为例子。我们准备一个简单的 module——hello.py：

```
[hello.py]
a = 1
b = 2
```

在这个简单的 module 中，我们仅仅创建了两个符号，a 和 b。图 14-4 显示了 Python 对 hello module 的 import 动作将对 Python 产生怎样的影响。

```
>>> import sys
>>> def containHello():
        return 'hello' in sys.modules.keys()

>>> containHello()
False
>>> import hello
>>> containHello()
True
>>> dir()
['__builtins__', '__doc__', '__name__', 'containHello', 'hello', 'sys']
>>> id(hello)
13861136
>>> id(sys.modules['hello'])
13861136
>>> type(hello)
<type 'module'>
```

图 14-4 探索对 hello module 的加载

操作 type() 的结果显示，import 机制确实创建了一个新的 module。而令人惊奇的是，Python 对 hello module 进行 import 操作的结果不仅将 hello module 引入到了当前的 local 名字空间中，而且这个被动态加载的 module 也在 sys.modules 中拥有了一席容身之地。更进一步，从两个 id() 操作的结果看来，local 名字空间中的符号“hello”和 sys.modules 中的符号“hello”背后隐藏的其实是同一个 PyModuleObject 对象。

探索“hello”这个符号背后隐藏的这个 PyModuleObject 对象也非常地有意义，图

14-5 展示了对 hello 内部的探测。

```
>>> dir(hello)
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'b']
>>> print hello.__dict__.keys()
['a', 'b', '__builtins__', '__file__', '__name__', '__doc__']
>>> print hello.__name__
hello
>>> print hello.__file__
C:\Python25\Lib\idlelib\hello.py
```

图 14-5 探测 hello module 的内部信息

这里可以看到，module 对象内部实际上是通过一个 dict 在维护所有的（属性，属性值）。对，说白了，同 class 一样，module 又是一个名字空间。从 `PyModuleObject` 结构的定义我们也能清楚地看到这一点，在 module 内部，有一些关于 module 的元信息，比如 module 的名字，module 所容身的文件名。

如果这时你查看 hello.py 所在的目录，这里又有一个令人惊奇的发现，我们会看到 hello.py 文件经过 Python 编译后存储编译结果的 hello.pyc 文件，由此可见，Python 在 import 的过程中，对 hello.py 暗中下了手，生成了 hello.py 的编译结果 hello.pyc。

在 hello module 中，还有一个奇怪的符号，`__builtins__`，如果你还记得，在 Python 初始化完成之后，我们敲入 `dir()`，显示的结果中同样有一个"`__builtins__`"符号，这两个符号有什么关系呢？图 14-6 展示了我们对这个疑问的探测。

```
>>> dir()
['__builtins__', '__doc__', '__name__', 'containHello', 'hello', 'sys']
>>> dir(hello)
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'b']
>>> type(__builtins__)
<type 'module'>
>>> type(hello.__builtins__)
<type 'dict'>
>>> id(__builtins__)
11279856
>>> id(hello.__builtins__)
11286544
```

图 14-6 module 中的__builtins__与 local 名字空间中的__builtins__

原来当前的 `local` 名字空间中的"`__builtins__`"和 hello module 中的"`__builtins__`"虽然名字完全一样，但却是完全不同的家伙，一个是 module 对象，而另一个是 dict 对象，从两者的 `id` 也可看出，它们两个完全没有关系。

但是，等一等，它们真的没有关系吗？如果没有关系，Python 为什么会将相同的名字赋予它们，徒增迷惑？刚才我们提到，在 module 中，实际上是通过一个 dict 在维护属性的名字和值之间的关系，既然 `local` 名字空间中的"`__builtins__`"符号对应一个 module

对象，而 hello module 中的“__builtins__”符号对应一个 dict 对象，有没有可能……嗯，我们来探测一下，如图 14-7 所示。

```
>>> id(hello.__builtins__)
11286544
>>> id(__builtins__.__dict__)
11286544
>>> id(sys.modules['__builtin__'].__dict__)
11286544
```

图 14-7 module 中的__builtins__与 local 名字空间中的__builtins__间的联系

从图 14-7 中我们发现，果然，hello module 中的__builtins__符号对应的 dict 正是当前名字空间中__builtins__符号对应的 module 对象所维护的那个 dict 对象，而其实它们两个都只是表象，它们背后的真身实际上就是我们在对 Python 运行环境初始化分析中看到的那个__builtin__module 及它所维护的 dict，这个__builtin__module 和其他被 Python 预先加载进内存中的 module 一样，维护在 sys.modules 中。

## 14.2.2 嵌套 import

上面我们研究了对于单个独立的 module 的 import 动作，下面我们来探索一下嵌套的 import 动作。所谓嵌套的 import 动作，即是指当 Python 在执行“import A”时，在 A 这个 module（A.py）中又激活另一个 import 动作，比如“import B”。我们来看看这时会发生什么有趣的事情：

```
[usermodule1.py]
import usermodule2

[usermodule2.py]
import sys
```

在 usermodule1.py 中，我们进行了一次 import 动作，加载 usermodule2，而在 usermodule2.py 中，我们又一次激活了 import 机制，加载 sys module。首先，我们需要对 usermodule1 进行 import 动作，以推倒第一块多米诺骨牌。加载后的结果如图 14-8 所示。

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> import usermodule1
>>> dir()
['__builtins__', '__doc__', '__name__', 'usermodule1']
>>> dir(usermodule1)
['__builtins__', '__doc__', '__file__', '__name__', 'usermodule2']
>>> dir(usermodule1.usermodule2)
['__builtins__', '__doc__', '__file__', '__name__', 'sys']
```

图 14-8 在 module 中嵌套 import 操作

我们发现，在 usermodule1.py 和 usermodule2.py 中进行的 import 动作并没有影响到上一层的名字空间，而只是影响到了各个 module 自身的名字空间，更准确地说，是影响到了各个 module 自身所维护的那个 dict 对象。那么在 usermodule1.py 中加载的 usermodule2 是否会影响到全局的 module 集合，即 sys.modules 呢？图 14-9 显示，确实，全局 module 集合受到了影响。

```
>>> import sys
>>> sys.modules['usermodule2']
<module 'usermodule2' from 'C:\Python25\Lib\idlelib\usermodule2.py'>
```

图 14-9 sys.modules 中的 usermodule2

实际上，所有的 import 动作，不论是发生在什么时间、发生在什么地方，都会影响到全局 module 集合，这样做有一个好处，即如果程序的另一点再次 import 这个 module，Python 虚拟机只需要将全局 module 集合中缓存的那个 module 对象返回即可，如图 14-10 所示。

```
>>> import usermodule1
>>> import sys
>>> id(usermodule1.usermodule2)
13860912
>>> id(sys.modules['usermodule2'])
13860912
>>> import usermodule2
>>> id(usermodule2)
13860912
```

图 14-10 从系统 module 缓存中 import 同一个 module

再从 usermodule1 中接引入了 usermodule2 后，我们在当前的 local 名字空间中再一次直接引入 usermodule2，从 3 个 id 操作的结果可以看到，第二次直接引入 usermodule2 时，确实返回了已经被缓存在全局 module 集合中的 usermodule2。

## 14.2.3 import package

在 module 的基础上，Python 又提供了一种管理 module 的机制，这就是 package 机制。我们知道，逻辑上相关联的一些 class 应该经常聚合到一个 module 中，同样地，Python 提供的 package 机制与 module 机制是类似的，逻辑上相关联的一些 module 应该聚合到一个 package 中。如果说 module 是一种管理 class 函数的机制，那么 package 就是一种管理 module 的机制。当然，更进一步，多个较小的 package 又可以聚合成一个较大的 package，一个典型的例子是 Python 标准库中的 xml 的 package。在这个 package 中，多个 module、package 最终组织成了一个树形的结构，从而为最初散乱的 class 建立起了一种结构，通过这种结构，方便了类的管理和维护，也方便了用户的使用。图 14-11 显示了我们对 xml package 的模拟结构图。

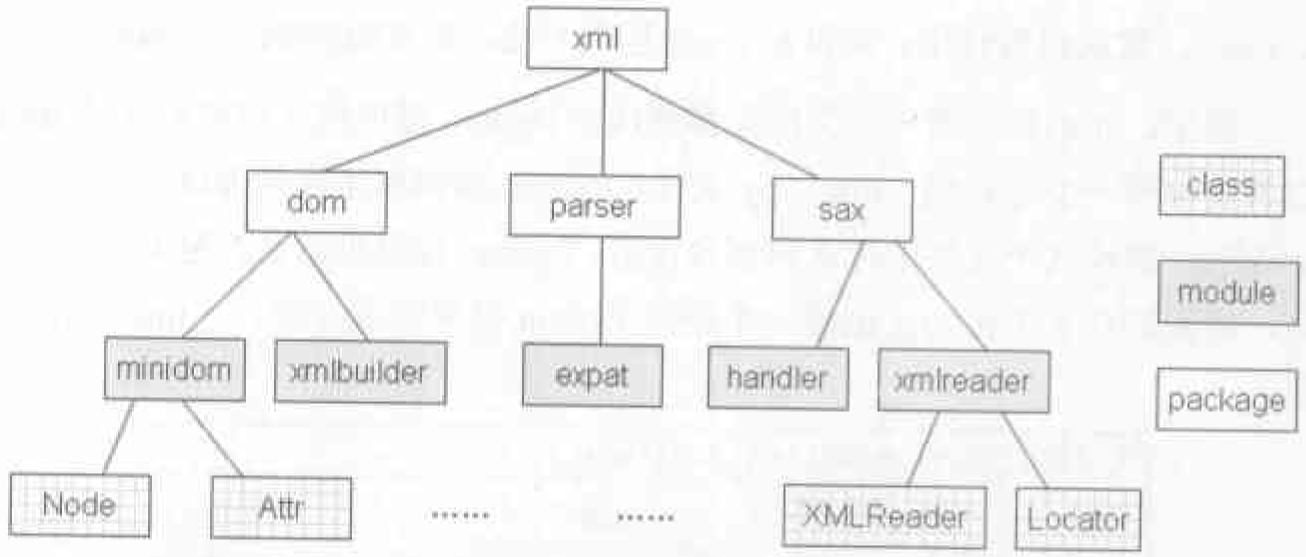



图 14-11 xml package 的结构

在 Python 中，module 是由一个单独的文件来实现的，可以是 py 文件，或者 pyc 文件，甚至是用 C 扩展的 dll 文件。而对于 package，Python 使用了文件夹来实现它，可以说，一个文件夹就是一个 package，里面容纳了一些 py、pyc 或 dll 文件，这种方式就是把 module 聚合成一个 package 的具体实现。

我们从一个最简单的例子开始对 package 的动态加载机制进行研究，首先，需要建立一个 package，很简单，就是创建一个文件夹，再在文件夹下创建一个 py 文件，如图 14-12 所示。

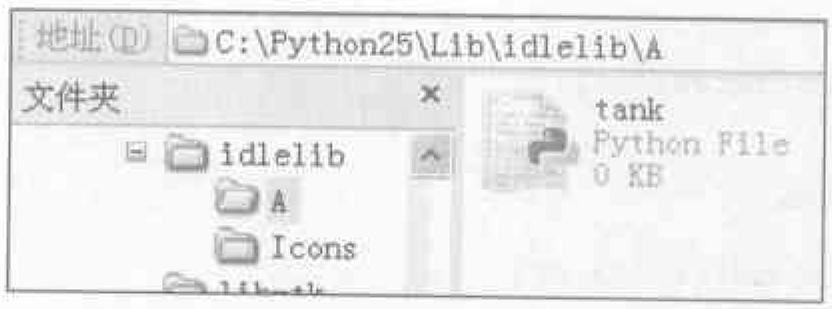



图 14-12 无效的 package 的目录结构

下面我们来对 package A 中的 tank module 进行 import 操作，如图 14-13 所示。

```
>>> import A.tank

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    import A.tank
ImportError: No module named A.tank
>>> import A

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    import A
ImportError: No module named A
```

图 14-13 对 package A 进行 import 动作

我们尝试了两种方式，结果 Python 虚拟机都不认账，抛出了异常，说是找不到 A 或

A.tank。这就很奇怪了，A 和 A.tank 远在天边，近在眼前啊，Python 怎么能找不到呢？

原来，并不是任何一个文件夹都能成为 Python 虚拟机认可的合法的 package，只有在文件夹中有一个特殊的__init__.py 文件，Python 虚拟机才会认为这个文件夹是一个合法的 package。即使这个文件中什么内容也没有，Python 虚拟机也认为这是一个合法的 package。好，那我们在 A 中再为 A 创建一个通向 Python 世界的通行证：__init__.py，如图 14-14 所示。

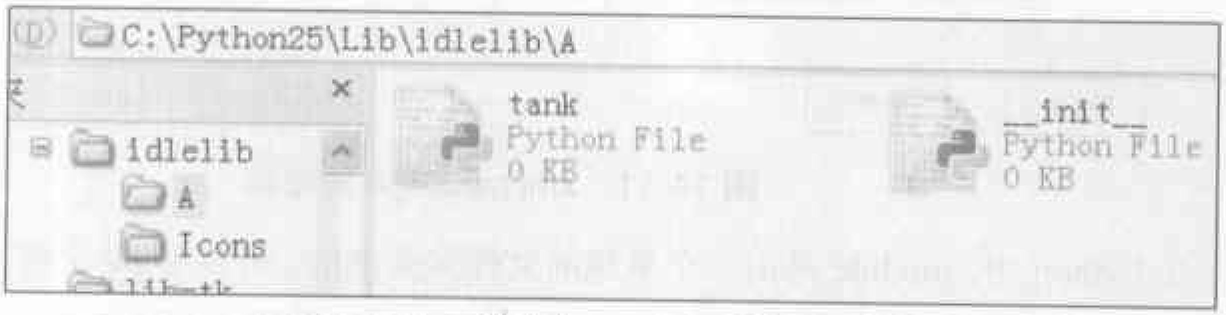



图 14-14 合法的 package 的目录结构

好了，现在我们重新试一试动态加载机制：

```
>>> dir()
['A', '__builtins__', '__doc__', '__name__']
>>> dir(A)
['__builtins__', '__doc__', '__file__', '__name__', '__path__', 'tank']
>>>
>>> import sys
>>> id(A)
13861008
>>> id(sys.modules['A'])
13861008
>>>
>>> id(A.tank)
13861104
>>> id(sys.modules['tank'])

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    id(sys.modules['tank'])
KeyError: 'tank'
>>> id(sys.modules['A.tank'])
13861104
```

图 14-15 对 package A 的成功加载

在图 14-15 中我们看到，这一次的动态加载机制确实成功了。在图 14-15 显示的结果中，有一些有趣的现象。虽然我们加载的只是 A 中的 tank module，但实际上却连 A 也一起加载进来了，可以看到，在 Python 中，module 和 package 之间的区别实际并不是那么僵硬，package 也可以像 module 一样被加载，行为和 module 其实是一样的，所以在 sys.modules 这个 Python 系统的 module 集中营中，我们也看到了 A 的身影。

对于 tank 的访问必须通过 A.tank 来实现，这样有一个好处，比如在 package B 中也有一个 tank module，那么这两个符号都能加载进来，通过 A.tank 和 B.tank 分别进行

引用，不会产生名字冲突，这一点，与 C++中的 namespace 机制和 Java 中的 package 机制是一样的。有趣的是，在 sys.modules 中，却没有名为“tank”的 module，只有名为“A.tank”的 module，仔细想一想，确实应该如此，否则 A.tank 和 B.tank 在 sys.modules 中如何能和平共存呢☺。

对于为何会在加载 A.tank 的同时也将 A 加载，可能我们会有些疑问，毕竟，这可能并不是用户期望的行为，但是对于 Python 而言，这是必须的。前面提到，对于 tank module 的引用只能通过 A.tank 来实现，根据我们了解的 Python 虚拟机的运作机理，Python 会首先在当前的 local 名字空间中查找符号“A”对应的对象 objA，然后再在 objA 的属性集合（名字空间）中查找符号“tank”。所以如果不将 A 也加载进来，则当前名字空间中就无法搜索到符号“A”，而对 tank module 的引用也无从谈起。

同时，加载 A 还有另外一个好处，如图 14-16 所示的例子。

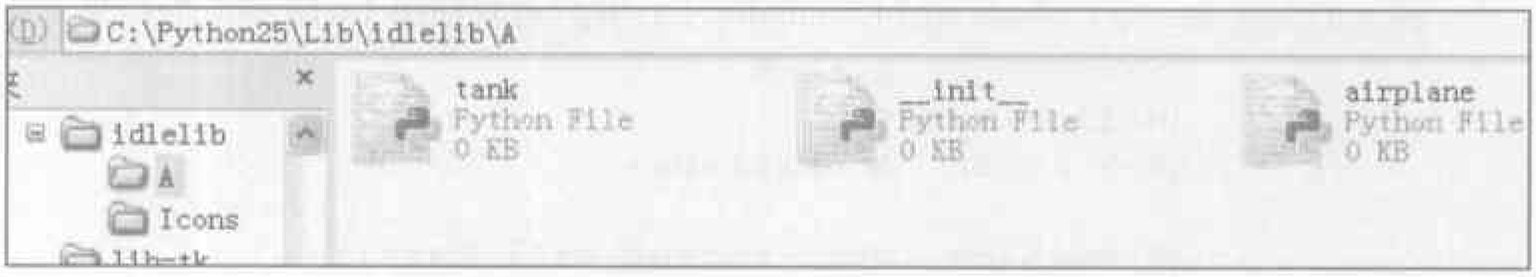



图 14-16 合法的 package 的目录结构

在 package A 中，有两个 module：tank 和 airplane，并且，在__init__.py 中，我们加入了一条打印“Hello Import A”的语句，通过依次加载这两个 module，来观察 Python 的行为，如图 14-17 所示。

```
>>> import A.tank
Hello Import A
>>> dir(A)
['__builtins__', '__doc__', '__file__', '__name__', '__path__', 'tank']
>>>
>>> import A.airplane
>>> dir(A)
['__builtins__', '__doc__', '__file__', '__name__', '__path__', 'airplane', 'tank']
>>> A.__path__
['C:\\Python25\\Lib\\idlelib\\A']
>>>
```

图 14-17 对 package 中 module 的独立加载

在动态加载 tank module 时，虽然会连带将 A 也加载，但是并没有加载 airplane，这是用户期望的行为。再次加载 aireplane module 时，我们发现，并没有再次加载 A 了（“Hello Import A”只输出了一次）。这样的现象是由 Python 对 package 中的 module 的动态加载机制的实现决定的。

在加载 package 下的 module 时，比如 A.B.c，Python 内部将这个 module 的表示视为一个树状的结构，即 B 位于节点 A 下，而 c 位于节点 B 下。Python 虚拟机在进行动态加

载时，首先会将这个 module 的树状结构分解，形成（A，B，c）这样的节点集合，然后从左到右依次去 sys.modules 中查找每一个符号所对应的 module 是否已经被加载了。如果是一个 package 被加载了，比如说 A 被加载了，那么在 A 对应的 PyModuleObject 对象中维护着一个元信息__path__，表示这个 package 的路径。如果当前 A 被加载到 sys.modules 中了，而 B 和 c 还没有，那么接下来对 B 的搜索将只在 A.__path__中进行，而不在 python 的所有搜索路径中执行了。

对应图 14-17 所示的例子，在加载了 A.tank 之后，A 已经被加载到了 sys.module 中，所以在加载 A.airplane 时，Python 虚拟机首先在 sys.modules 中搜索符号“A”对应的 PyModuleObject 对象，发现其已经被加载，则不会再次进行加载，所以我们没有看到“Hello Import A”的输出，而且 Python 虚拟机发现其元信息__path__为 C:\Python25\Lib\idlelib\A，那么对 airplane 的搜索就只在该路径下进行。如果我们将 airplane 移动到 A 的父目录，即 C:\Python25\Lib\idlelib 中，我们看看如图 14-18 所示的结果。

```
>>> import A.tank
Hello Import A
>>> import A.airplane

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    import A.airplane
ImportError: No module named airplane
>>> import airplane
>>> airplane.__file__
'C:\\Python25\\Lib\\idlelib\\airplane.py'
```

图 14-18 加载 A.aireplane 失败

非常清晰的结果，虽然对 Python 虚拟机而言，通过 import airplane、airplane 实际上是可以访问到的，这表明 Python 虚拟机在它的某条 module 搜索路径中是能够成功搜索到 airplane 的，但是 A.airplane 限制了 Python 虚拟机对 airplane 的搜索范围，所以动态加载的动作失败，导致异常抛出。

### 14.2.4 from 与 import

在 Python 的 import 机制中，有一种精确控制所加载的对象的方法，通过 from 关键字与 import 的结合，我们可以只将我们期望的 module，甚至是 module 中的某个符号，动态加载到内存中。这种机制使得 Python 虚拟机在当前名字空间中引入的符号可以尽可能地少，从而更好地避免名字空间遭到污染。

参考如图 14-14 所示的 package 布局，假如我们希望动态加载 tank module，在这之

前，我们讨论了一种方法，即：import A.tank。从图 14-15 所示的结果中可以看到，在当前的 local 名字空间引入了 A，而我们希望的符号“tank”则在 A 的属性中，不在当前的 local 名字空间中。

显然，我们希望更方便一些，即能够直接将符号“tank”及其对应的 module 引入到当前的 local 名字空间中，通过 from 和 import 的联手，能够完美地完成这个任务：“from A import tank”。图 14-19 展示了使用 from 的结果。

```
>>> from A import tank
Hello Import A
>>> dir()
['__builtins__', '__doc__', '__name__', 'tank']
>>>
>>> import sys
>>> sys.modules['tank']

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    sys.modules['tank']
KeyError: 'tank'
>>> sys.modules['A']
<module 'A' from 'C:\Python25\Lib\idlelib\A\__init__.pyc'>
>>> sys.modules['A.tank']
<module 'A.tank' from 'C:\Python25\Lib\idlelib\A\tank.pyc'>
```

图 14-19　直接将 package 中的 module 引入当前 local 名字空间

果然，Python 虚拟机直接将符号“tank”引入到了当前名字空间，但是当进行了更深一层的探索之后，我们发现，其实这种方式的本质是与 import A.tank 一样的，都是将 package A 和 module A.tank 动态加载到了 Python 的 sys.modules 集合中，不同之处只是在于当 import 的动作要结束时，Python 会在当前的 local 名字空间中引入什么符号。在 import A.tank 中，Python 虚拟机引入了符号“A”，并将其映射到 module A；而在 from A import tank 中，Python 虚拟机则引入了符号“tank”，并将其映射到了 module A.tank。

对于 from 和 import 的结合，还有一种更精妙的用法，即仅仅将某个 module 中的一些对象暴露到当前名字空间中，在这之前的所有例子中，我们要么不加载，要加载就将整个 module 都加载并引入到当前名字空间中。而 from 与 import 的结合则能精确地操纵 module 中的某个部分，我们来见识一下，图 14-20 显示了动态加载语句：from A.tank import a 的结果，其中 a 是 A.tank 中的一个整数对象。

```
>>> from A.tank import a
Hello Import A
>>> dir()
['__builtins__', '__doc__', '__name__', 'a']
>>> a
1
>>>
>>> import sys
>>> sys.modules['A']
<module 'A' from 'C:\Python25\Lib\idlelib\A\__init__.pyc'>
>>> sys.modules['tank']

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    sys.modules['tank']
KeyError: 'tank'
>>> sys.modules['A.tank']
<module 'A.tank' from 'C:\Python25\Lib\idlelib\A\tank.py'>
>>> sys.modules['A.tank.a']

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    sys.modules['A.tank.a']
KeyError: 'A.tank.a'
```

图 14-20 精确控制对 package 下某个 module 中某个符号的引入

从 from A.tank import a 的输出结果我们其实已经可以看到一些线索了，看上去，package A 已经被加载到了内存中，同样我们可以推测，和 import A.tank、from A import tank 一样，A.tank 也被加载到了内存中，在后面的操作结果中我们清晰地看到了这一点。

我们还看到，所引入的符号“a”确实只是 A.tank module 中的一个对象，并没有 A.tank.a 这样的 module 存在。

最后，Python 还为我们提供了一种机制，允许将一个 module 中的所有对象一次性地引入到当前名字空间中，如图 14-21 所示。

```
>>> from A.tank import *
Hello Import A
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'b']
```

图 14-21 引入 package 下 module 中的所有符号

## 14.2.5 符号重命名

在前面的讨论中，我们关注的焦点实际上是“Python 虚拟机将哪个 module 动态加载

进了内存"。而对于动态加载机制，Python 还提供了一种符号重命名的机制，这个机制为动态加载机制提供了更大的灵活性，实际上我们可以认为这种重命名机制关注的焦点是"Python 虚拟机将 module 以怎样的形式暴露给当前名字空间"。

之前的例子都是将 module 自身的名字暴露到了当前的 local 名字空间，Python 通过 as 关键字可以控制 module 以什么名字被引入到当前的 local 名字空间中，如图 14-22 所示。

```
>>> import A.tank as Tank
Hello Import A
>>> dir()
['Tank', '__builtins__', '__doc__', '__name__']
>>> import sys
>>> sys.modules['Tank']

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    sys.modules['Tank']
KeyError: 'Tank'
>>> sys.modules['A.tank']
<module 'A.tank' from 'C:\Python25\Lib\idlelib\A\tank.pyc'>
```

图 14-22 更换引入到当前 local 名字空间中的 module 名

第一眼看见图 14-22，我们就心中有数了，A 和 A.tank 已经被加载到了 Python 的 module 集合中，有兴趣的朋友可以自己验证。令人感兴趣的是，在当前名字空间中果然出现了通过 as 控制的符号"Tank"，而且 Tank 实际上被映射到了 module A.tank，这一点与图 14-15 所示的结果有很大不同，在图 14-15 中，我们看到了符号"A"被引入到了当前名字空间，而在图 14-22 中，符号"A"已经消失了，取而代之的是我们指定的符号"Tank"，如此一来，虽然在 sys.modules 中还有 A 存在，但是访问 A 已经是不可能了。仅仅加了一个关键字 as，就取得了这样戏剧性的改变，原因究竟是什么呢？呃，列位看官，且听以后分解☺。

## 14.2.6 符号的销毁与重载

为了使用一个 module，无论是 Python 的 standard module 还是用户自己的 module，都需要通过 import 将其动态加载到内存中，在使用了这个 module 之后，可能我们需要将其删除。为什么需要将其删除呢？通常原因很多，也许是想释放该 module 占用的内存，也许是想避免名字空间过于庞大，以保证在名字空间中搜索某个符号的动作的效率不会成为 Python 运行时的瓶颈，如此等等。通常，我们以为 del 能够很好地完成工作，毕竟，从一开始接触 python 我们就知道，当你需要删除一个对象时，del 吧。我们总是直觉地认为这样一刀下去，天下太平，毕竟"del"这个名字总不能是白叫的吧。

但是且慢，这样的动作真的能保证 module 被销毁吗？或者更准确地说，“符号的销毁”和“符号关联的对象的销毁”是一个概念吗？我们已经看到 Python 向我们隐藏了太多的动作，Python 采取了太多的缓存策略，当然，对于 Python 的使用者来说，隐藏这些复杂性是好事，但是当我们希望彻底了解 Python 的行为时，就必须将这些隐藏的东西挖出来。图 14-23 展示了我们的挖掘结果。

```
>>> import A.tank as Tank
Hello Import A
>>> dir()
['Tank', '__builtins__', '__doc__', '__name__']
>>> del Tank
>>> dir()
['__builtins__', '__doc__', '__name__']
>>>
>>> import sys
>>> sys.modules['A.tank']
<module 'A.tank' from 'C:\Python25\Lib\idlelib\A\tank.pyc'>
>>> id(sys.modules['A.tank'])
13860912
>>>
>>> import A.tank as Tank
>>> dir()
['Tank', '__builtins__', '__doc__', '__name__', 'sys']
>>> id(Tank)
13860912
```

图 14-23 考察对符号的 del 操作

在 del 之后，符号“Tank”确实从当前名字空间中消失了，但是我们随后看到，消失的仅仅是“Tank”这个符号，而其背后隐约闪现的 module A.tank 依然在 Python 的 module 缓存中，巍然不动。然而，尽管它还存在于 Python 系统中，但是我们的程序再也无法访问到这个 module，仿佛天地间再也不存在 A.tank 这个 module 一样。“存在即是被感知”，这个英国大主教贝克莱所提出的命题，这个经常被唯物主义者嘲弄的命题，对于 Python 中的 A.tank 而言，却是不折不扣的真理。我们的程序已经没有办法感知到 A.tank 的存在，Python 系统成功地向我们隐藏了这一切，所以，我们的程序认为：A.tank 已经不存在了。

为什么 Python 要采用这种看上去类似 module pool 的缓存机制呢，一个重要的原因是组成一个完整系统的多个 py 文件可能都会对某个 module 进行 import 动作，希望使用这个 module 所提供的功能。到了这里，我们看到，从 Python 的角度看，import 其实并不完全等同于我们所熟知的“动态加载”这个概念，它的真实含义是希望某个 module 能够被感知，即是**将这个 module 以某个符号的形式引入到某个名字空间**。如果 import 等于动态加载，那么 Python 将对同一个 module 执行多次动态加载，并且在内存中保存一个 module

的多个映像，这显然是愚蠢的。

所以 Python 引入了全局的 module 集合——`sys.modules`，这个集合作为 module pool，保存了 module 的唯一映像，当某个.py 文件通过 `import` 声明希望感知到某个 module 时，Python 将在这个 pool 中查找，如果该 module 已经存在于 pool 中，则引入一个符号到该.py 文件的名字空间中，并将其关联到该 module，使该 module 透过这个符号能够被.py 文件感知到；而如果该 module 不在 pool 中，这时 Python 才执行动态加载的动作。

难道这意味着一个 module 如果被加载之后就再也不能被改变？假如在加载了 module A 之后，我们更新了 A 中某个功能的实现，以提供更好的效率，我们希望我们的应用程序能使用新的 A module 中效率更高的某个功能的实现，难道 Python 就束手无策了？Python 的动态特性显然不会技止于此，它提供了一种重新加载的机制，这种机制是通过 builtin module 中的 reload 操作实现的。图 14-24 显示了重载的情形。

```
>>> import sys
>>> import robert
>>> dir(sys.modules['robert'])
['__builtins__', '__doc__', '__file__', '__name__', 'a']
>>> id(robert)
13836432
>>>
>>> # [1] : 添加代码'b = 2'到robert.py中
>>> reload(robert)
<module 'robert' from 'C:\Python25\Lib\idlelib\robert.py'>
>>> dir(sys.modules['robert'])
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'b']
>>> id(robert)
13836432
>>>
>>> # [2] : 删除'b = 2', 并添加代码'c = 3'到robert.py中
>>> reload(robert)
<module 'robert' from 'C:\Python25\Lib\idlelib\robert.py'>
>>> dir(sys.modules['robert'])
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'b', 'c']
>>> id(robert)
13836432
>>> robert.b
2
```

图 14-24 Python 中的 module 的重新加载机制

在图 14-24 的[1]处，我们在 robert.py 中添加了“`b = 2`”这个表达式，调用 `reload` 操作进行重新加载后，我们发现，在 `sys.modules` 中的 robert module 的确更新了。但是从 `id()` 操作的结果可以看到，Python 虚拟机并没有创建新的 module 对象。所以我们可以猜测，在 `reload` 背后，Python 只是在原有的 `robert module` 中添加了符号“`b`”及其对应的值。

接着，一个奇怪的现象出现在了图 14-24 的[2]之后。在[2]处，我们从 robert.py 中将

“b = 2”删除，同时添加了“c = 3”，按照我们的推测，这时 robert.py 中的属性应该是（a，c）才对，但是，白纸黑字，那个本应该被删除的 b 还是出现在了 robert module 的属性列表中，更奇怪的是，robert.b 的值还是以前的 2。这似乎预示着 Python 虚拟机在调用 reload()操作更新 module 时，只是将新的符号加入到 module 中，而不管 module 中的符号是否已经在源文件中被删除了。

这个猜测到底对不对呢？别急，这个谜底在本章的最后将被揭开。现在，我们看了太多的黑盒探测，已经迫不及待想要把这个黑盒撬开，看一看里面到底是如何运作的。OK，follow me，我们马上进入对 import 机制的源码剖析。

## 14.3 import 机制的实现

从前面的黑盒探测我们已经对 import 机制有了一个非常清晰的认识，Python 的 import 机制基本上可以切分成 3 个不同的功能：

- Python 运行时的全局 module pool 的维护和搜索；
- 解析与搜索 module 路径的树状结构；
- 对不同文件格式的 module 的动态加载机制。

完成了这 3 个功能，实际上我们自己也能实现一个与 Python 兼容的动态加载机制了。在这一节中，我们将深入 Python 的源码，仔细考察 Python 是如何实现这 3 个功能，将对 import 机制的认识从整体框架的基础上更深一层，到达代码的层次。

从前面的分析我们看到，尽管 Python 中 import 的表现形式千变万化，但是归根结底，都可以归结为：import x.y.z 的形式。因为对于 import sys 中的 sys 可是视为 x.y.z 的一种特殊形式；而诸如 from、as 与 import 的结合，实际上同样会进行 import x.y.z 的动作，只是最后在当前名字空间中引入符号时各有不同。所以我们对代码的分析将以 import x.y.z 这样的形式作为默认的 import 动作。

在本章第 1 节中我们看到，Python 的 import 机制的起点是 builtin module 中的 __import__操作，也就是 builtin___import__函数，我们的征途就从这里开始。

```
[bltinmodule.c]
static PyObject * builtin___import__(PyObject *self, PyObject *args,
  PyObject *kwds)
{
    static char *kwlist[] = {"name", "globals", "locals", "fromlist",
              "level", 0};
    char *name;
    PyObject *globals = NULL;
    PyObject *locals = NULL;
```

```
    PyObject *fromlist = NULL;
    int level = -1;
    //从 tuple 中解析出需要的信息
    if (!PyArg_ParseTupleAndKeywords(args, kwds, "s|OOOi:__import__",
            kwlist, &name, &globals, &locals, &fromlist, &level))
        return NULL;
    return PyImport_ImportModuleLevel(name, globals, locals, fromlist,
        level);
}
```

这里的 PyArg_ParseTupleAndKeywords 函数在 Python 自身的实现中是一个被广泛使用的函数，其原型如下：

```
int PyArg_ParseTupleAndKeywords( PyObject *args, PyObject *kw,
                                 const char *format, char *keywords[], ...)
```

这个函数的目的是将 args 和 kwds 中所包含的所有对象按 format 中指定的格式解析成各种目标对象。目标对象可以是 Python 中的对象，比如 PyDictObject 对象，PyStringObject 对象，也可能是 C 的原生类型，比如 int，char*等。

我们知道，这里的这个 args 实际上是一个 PyTupleObject 对象，包含了 builtin___import__函数运行所需要的所有参数和信息，它是 Python 虚拟机在执行 IMPORT_NAME 指令时打包而产生的，到了这里，Python 虚拟机进行了一个逆动作，即将打包后得到的这个 PyTupleObject 拆开，重新获得当初的参数。Python 在自身的实现中大量运用了这样的打包、拆包的策略，使得可变数量的对象能够很容易地在函数之间传递。

在解析参数的过程中，指定解析格式的 format 中可用的格式字符非常多，想要详细了解其所支持的所有格式字符可以参考 Python 自身所携带的文档。这里简要介绍一下 builtin___import__用到的格式字符。

其中，s 代表目标对象是一个 char*，通常用来将 tuple 中的 PyStringObject 对象解析成 char*；i 则用来将 tuple 中的 PyIntObject 对象解析为 int 类型的值；而 O 则代表解析的目标对象依然是一个 Python 中的合法对象，通常，这表示 PyArg_ParseTupleAndKeywords 不进行任何的解析和转换，因为在 PyTupleObject 对象中存放的肯定是一个 Python 的合法对象。

至于“|”和“:”，则非格式字符，而是指示字符，“|”指示其后所带的格式字符是可选的。也就是说，如果 args 中只有一个对象，那么 builtin___import__对 PyArg_ParseTupleAndKeywords 的调用也不会失败。其中，args 中的那个对象会按照“s”的指示被解析为 char*，而剩下的 global、local、fromlist 则将会按照“O”的指示被初始化为 Py_None，level 则保持不变。最后的那个“:”指示格式字符到此就结束了，其后所带字符串用于在解析过程中出错时输出错误信息时使用，可以看到，输出了“:”后面的那个“__import__”，就能很好地定位错误的出现位置了。

在完成了对参数的拆包动作之后，Python 进入了 PyImport_ImportModuleLevel:

```
[import.c]
PyObject *
PyImport_ImportModuleLevel(char *name, PyObject *globals, PyObject
  *locals, PyObject *fromlist, int level)
{
    PyObject *result;
    lock_import();
    result = import_module_level(name, globals, locals, fromlist, level);
    unlock_import();
    return result;
}
```

Python 虚拟机在进行 import 之前，会对 import 这个动作上锁，这样做是为了同步不同的线程对同一个 module 的 import 动作，如果没有这个同步，则会产生一些异常现象。在进行了 import 动作之后，还会通过 unlock_import 解开锁。我们关注的焦点在 import 动作，所以不去深究锁机制，直接进入 import_module_level（见代码清单 14-3）。

代码清单 14-3

```
[import.c]
static PyObject *
import_module_level(char *name, PyObject *globals, PyObject *locals,
                    PyObject *fromlist, int level)
{
    char buf[MAXPATHLEN+1];
    int buflen = 0;
    PyObject *parent, *head, *next, *tail;
    //[1]: 获得 import 动作发生的 package 环境
    parent = get_parent(globals, buf, &buflen, level);
    //[2]: 解析 module 的 "路径" 结构, 依次加载每一个 package/module
    head = load_next(parent, Py_None, &name, buf, &buflen);
    tail = head;
    while (name) {
        next = load_next(tail, tail, &name, buf, &buflen);
        tail = next;
    }
    //[3]: 处理 from *** import *** 语句
    if (fromlist != NULL) {
        if (fromlist == Py_None || !PyObject_IsTrue(fromlist))
            fromlist = NULL;
    }
    //import 的形式不是 from *** import ***, 返回 head
    if (fromlist == NULL) {
        return head;
    }

    //import 的形式是 from *** import ***, 返回 tail
    if (!ensure_fromlist(tail, fromlist, buf, buflen, 0)) {
        return NULL;
    }
    return tail;
}
```

我们回忆一下本章第 1 节中列出的 IMPORT_NAME 指令的实现代码。

```
[IMPORT_NAME]
        ……
        x = PyEval_CallObject(x, w);
        SET_TOP(x);
```

最终获得 x 就是这里 import_module_level 的返回值，可以看到，返回值依赖于 fromlist 这个参数，那么这个参数代表的是什么东西呢？

一般情况下，这个 fromlist 都是 Py_None，但是当 Python 虚拟机进行"from a import b, c"这样的动作时，fromlist 就不再无效了，而是成为一个诸如(b, c)这样的 PyTupleObject 对象。这个 fromlist 将如何影响 import 的行为，在以后的章节中将详细剖析。

## 14.3.1 解析 module/package 树状结构

在第 2 节的分析中我们已经发现，Python 对 x.y.z 的 import 动作实际上是沿着树状结构一层一层地展开的，也就是说，可以看作是对树状结构的遍历操作。更形象一些，我们可以将 x.y.z 看作是对一个二元树的遍历的轨迹，其中在遍历的过程中，我们对每个节点都只访问其右子树。对于这样的一个从根到叶节点的遍历操作，在学习数据结构时，就应该是滚瓜烂熟了，如下面的伪代码所示：

```
Void travel_tree(tree)
{
    parent = get_parent(tree);
    head = get_right_child(parent);
    tail = head;
    while(tail != NULL)
    {
        head = get_right_child(tail);
        tail = head;
    }
}
```

怎么样？是不是觉得跟 import_module_level 非常地神似？没错，import_module_level 的主要动作正是实现了对 x.y.z 这样的树状结构的遍历。在后面的描述中我们会越来越清晰地看到这一点。首先我们来看看 import_module_level 中 get_parent 完成的动作（见代码清单 14-4）。

**代码清单 14-4**

```
[import.c]
static PyObject* get_parent(PyObject *globals, char *buf, Py_ssize_t
  *p_buflen, int level)
{
    PyObject *namestr = NULL;
    PyObject *pathstr = NULL;
    PyObject *modname, *modpath, *modules, *parent;
```

```
    //[1]: 获得当前module的名字
    namestr = PyString_InternFromString("__name__");
    pathstr = PyString_InternFromString("__path__");
    *buf = '\0';
    *p_buflen = 0;
    modname = PyDict_GetItem(globals, namestr);
    modpath = PyDict_GetItem(globals, pathstr);
    if (modpath != NULL) {
            //[2] : 在package的__init__.py中进行import动作
            Py_ssize_t len = PyString_GET_SIZE(modname);
            strcpy(buf, PyString_AS_STRING(modname));
    }
    else {
        //[3] : 在package的module中进行import动作
        char *start = PyString_AS_STRING(modname);
        char *lastdot = strrchr(start, '.');
        size_t len;
        len = lastdot - start;
        strncpy(buf, start, len);
        buf[len] = '\0';
    }

    while (--level > 0) {
            char *dot = strrchr(buf, '.');
            *dot = '\0';
        }
        *p_buflen = len;

        //[4]: 在sys.modules中查找当前package的名字对应的module对象
        modules = PyImport_GetModuleDict();
        parent = PyDict_GetItemString(modules, buf);
        return parent;
}
```

其中的函数 level 一般情况下都为-1，这时，level 不对 get_parent 产生影响，所以我们这里不考虑 level 这个参数。

函数 get_parent 的功能是返回一个 package，这个 package 是当前的 import 动作执行的环境。举个例子吧，假设我们在如图 14-25 所示的目录结构中考察 import 动作。

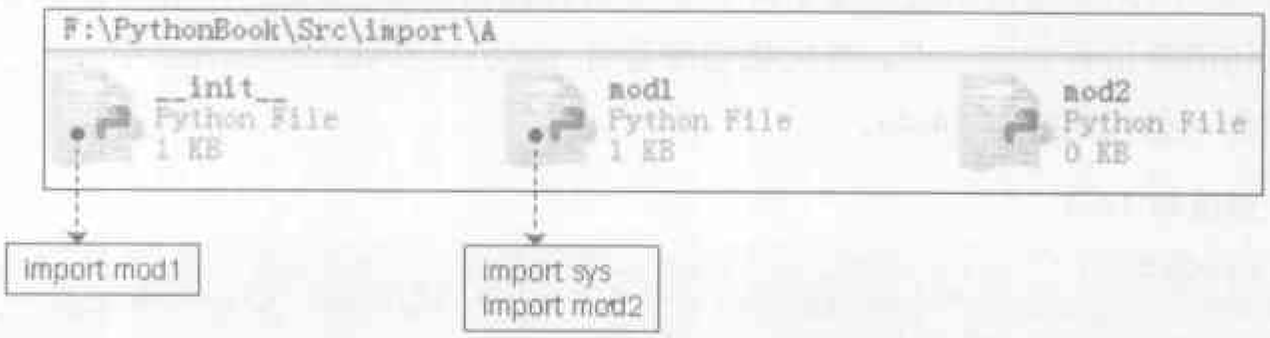



图 14-25 package A 的目录布局

现在有一个 package A，在 A 中，有名为 mod1 和 mod2 的两个 module。当 Python 虚

拟机执行“import A”时，会动态加载目录 A 下的__init__.py。现在，在__init__.py 中，Python 虚拟机又执行了“import mod1”，那么 Python 虚拟机认为，对于 mod1 这个 module 的 import 动作是发生在 package A 的环境中的。这个所谓的环境，实际上最重要的一点就是 package A 的元信息：__path__，即目录 A 的路径信息。因为一般来说，在 A/__init__py 中 import 一个 module（我们这里是 mod1），那么合理的行为是该 module 应该也在目录 A 中，或 Python 的标准搜索路径中，这样实际上也才能将 package 作为 module 的管理机制。

假如在 package A 中的 import 动作能影响到 package B 中的 mod1，那么如果 package A 和 B 中也同时有 mod1，那么 Python 究竟该选择哪一个作为 import 的对象呢？显然，package 是 import 的边界，import 动作不能跨越 package 进行，所以在 package 中进行 import 动作时，我们需要 package 的__path__元信息，以限制对 module 的搜索范围。

Python 虚拟机在执行“import A”时，会为 package A 创建一个 module 对象，同时会在该 module 维护的 dict 中添加两个表示元信息的属性：__name__和__path__。而 Python 虚拟机从 A/__init__.py 中执行“import mod1”时，也会为 mod1 创建一个 module 对象，同时也会设置__name__属性，但是这时就不设置__path__属性了。

统一一下，我们说，当 Python 虚拟机 import 一个 package 或 module 时，都会创建一个 module 对象，并且设置其__name__和__path__，只不过 module 对应的__path__为空罢了。对于我们的例子，A 对应的 module 对象中的属性集合为{“__name__”：“A”，“__path__”：“F:\PythonBook\Src\import\A”；而 mod1 对应的 module 对象中的属性集合为{“__name__”：“A.mod1”}。

当在 package A 的环境（包括__init__.py、mod1.py 和 mod2.py，但仅考虑__init__.py 和 mod1.py 就足够了）中执行 Python 代码时，**这时的 global 名字空间（globals）就将是__init__.py 或 mod1.py 对应的 module 对象所维护的** dict。代码清单 14-4 的[1]处从 globals 中获得__name__属性，即 A 或 A.mod1。随后，再根据 globals 名字空间中的__path__属性区分目前处理的 import 动作是在__init__.py 中发生的还是在 mod1.py 中发生的。代码清单 14-4 的[2]处处理 import 动作发生在__init__.py 中发生的情况；而代码清单 14-4 的[3]处处理 import 动作在 mod1.py 中发生的情况。不管是代码清单 14-4 的[2]还是代码清单 14-4 的[3]，其目的都是获得 package A 的名字“A”，并将其拷贝到 buf 中。

最后在代码清单 14-4 的[4]处，根据 buf 在 Python 的全局 module 集合中查找名为“A”的 module 对象，将搜索得到的对象返回。注意[4]处的 PyImport_GetModuleDict 即是获得全局 module 集合——sys.modules——的操作。

正是因为代码清单 14-4 的[3]将 module 的名字“A.mod1”退化为了 package 的名字“A”，所以，在 mod1.py 中执行的“import mod2”才能成功，因为只有 get_parent 返回了“A”

对应的 module 对象，接下来 Python 虚拟机才知道后续的 import 动作是发生在 package A 的环境下，寻找的 module 的路径应该基于 package A 的路径，只有基于 package A 的路径才能找到 mod2，因为它的路径为："A\mod2.py"。

到了现在，我们可以进行这样的一种抽象：Python 中的 import 动作都是发生在某一个 package 的环境下。那么如果 import 动作并不是真的发生在一个 package 中时，我们怎么来完善我们的抽象呢？就像我们在交互式环境下进行 import 动作时，我们并没有处于任何一个 package 中。

我们假设最初 Python 执行时，就是在一个系统默认的 package，__main__ package，中进行的，所以这时的 import 动作是发生在__main__ package 中的。所以，对应于这个__main__ package，get_parent 也需要返回一个与其对应的 module 对象。然而这个所谓的__main__ package 只是我们为了完善理论而创建的一个概念上的东西，并非一个真实的 package，所以我们强制与它对应的 module 对象是 Py_None，那么我们所需要的 pakcage 的__path__元信息呢，显然 Py_None 中没有这个信息，那么，就让 Python 默认的搜索路径成为这个__main__ package 的__path__元信息吧。

到此，我们将所有的 import 动作都归一到同一个抽象原则下：Python 中的 import 动作都是发生在某一个 package 的环境中。

注意我们说 import 动作是发生在一个 package 的环境中，而并非一个 module 的环境下，在这里，区分 package 和 module 十分重要。假如 Python 在__main__ package 下进行 import mod1 的动作，而在 mod1.py 中，又进行了 import mod2 的动作，那么这两个 import 动作都是发生在__main__ package 环境下的，尽管 import mod2 这个动作是由 mod1 触发的。这个约束是由我们在前面所剖析的代码清单 14-4 的[3]处所完成的动作保证的。

这样的区别看上去好像有些人工雕凿的痕迹，实际上，它源自 Python 中一个本质的抽象：package 是由 module 聚合而成。更清楚的表述是：module 属于一个 package。我们不能说，module1 属于 module2。我们前面已经看到，module 的路径实际上是一种树状结构，从图 14-11 中可以看到，在这个树状结构中，module 的父节点只能是 package，而不可能是另一个 module。

所以，对于 import mod1 和 import mod2，get_parent 都将返回__main__ package 对应的 module 对象：Py_None。

在获得了 import 动作执行的 package 环境之后，Python 虚拟机立即通过 load_next 开始了在 package 环境中对 module 的 import 动作见代码清单 14-5。

**代码清单 14-5**

```
[import.c]
static PyObject *
```

```
load_next(PyObject *mod, PyObject *altmod, char **p_name, char *buf, int
*p_buflen)
{
    char *name = *p_name;
    char *dot = strchr(name, '.');
    size_t len;
    char *p;
    PyObject *result;

    ......//[1]: 获得下一个需要加载的 package 或 module 的名字

    //[2]: 对 package 或 module 进行 import 动作
    result = import_submodule(mod, p, buf);
    if (result == Py_None && altmod != mod) {
        result = import_submodule(altmod, p, p);
    }
    return result;
}
```

在第 2 节的分析中我们已经看到，对于 import x.y.z 这样的形式，Python 采用了一种逐次加载的方法，也就是说，Python 将首先加载 package x，然后加载 package y，最后加载 module z。从图 14-11 就可以看出，在 x.y.z 这样的树状结构中，只有最后一个节点才能是 module。

这样的算法定义出一种序结构，假如在这个序结构上定义一个名为 next 的函数，那么以 x 为参数，next 函数的结果是 y，以 y 为参数，next 函数的结果是 z，而 x 同样也是 next 函数的结果。为了从 next 函数得到 x，我们需要有一个虚拟的起始节点，这样，module 的位置结构最终被抽象成了__main__.x.y.z 的形势。虽然我们说__main__是一个虚拟的起始节点，但在 Python 中，这个名为__main__的 package 实际上是有意义的，它的__path__元信息就是 Python 默认的搜索路径。

在 Python 的 import 机制中，load_next 正是 next 函数的实现，在 load_next 函数中，Python 首先将获得下一个需要动态加载的 package 或 module，这一部分在代码清单 14-5 的[1]处完成，由于这部分代码比较繁琐，所以这里没有详细列出。

在获得了需要加载的 package 或 module 的名字——设为 m——之后，load_next 调用 import_submodule 对 m 进行 import 动作，注意这里的 import 并不一定就意味着动态加载，前面我们说了，Python 还维护着一个 module pool 呢。

在 load_next 函数的参数中，p_name 和 buf 都是由 import_module_level 中传入，*p_name 中存储当前还剩下的 module 的数状结构，而 buf 中将存储已经完成 import 动作的树状结构，比如对于 import xml.sax.xmlreader，在整个逐次 import 的过程中，将会按照 import 的进行次序出现如下 4 种(*p_name, buf)组合：("xml.sax.xmlreader", ""), ("sax.xmlreader", "mxl"), ("xmlreader", "xml.sax"), (NULL, "xml.sax.

xmlreader"）。当*p_name 中的字符串变为 NULL 时，也就意味着 import 已经完成了，这时 buf 中将存储整个 module 路径的树形结构。在 import_module_level 中，整个逐次 import 的过程被清晰地展示了出来。

```
[import.c]
static PyObject*
import_module_level(char *name, PyObject *globals, PyObject *locals,
PyObject *fromlist)
{
    ......
    head = load_next(parent, Py_None, &name, buf, &buflen);//altmod == Py_None

    tail = head;
    while (name) {
        //altmod == mod == tail
        next = load_next(tail, tail, &name, buf, &buflen);
        tail = next;
    }
    ......
}
```

每一次调用调用 load_next 之后，完成一个部分（package 或 module）的 import，并且变换(*p_name, buf)组合，当整个 import 动作完成之后，*p_name，也就是 import_module_level 中的 name，将成为 NULL，这时 import_module_level 将结束这个逐次 import 的动作。

## 14.3.2 加载 module/pakcage

在 load_next 中调用的 import_submodule 完成了搜索 module 并加载 module 的工作，其原形如下：

```
static PyObject* import_submodule(PyObject *mod, char *subname, char
  *fullname)
```

其中，第一个参数 mod 就是我们在讨论函数 get_parent 时提到的那个 import 动作的执行环境，import_submodule 需要其中的__path__元信息来完成搜索 module 的任务，第二个参数和第三个参数可能有些使人迷惑。以 import xml.sax.xmlreader 为例，在最终 import xmlreader 时，subname 就是 xmlreader，而 fullname 为 xml.sax.xmlreader。这涉及 import_submodule 的两个任务：搜索 module 和维护全局 module 集合 sys.modules。当 import_submodule 执行时，Python 虚拟机在 mod 的__path__元信息提供的路径中搜索 module，需要搜索的是名为 subname 的文件；而当 import_submodule 完成了动态加载，需要将新创建的 module 加入到 Python 的 module pool 中时，还记得上面第 2 节中的分析吗？并不存在名为"xmlreader"的 module，在 module pool 中，只有名为"xml.sax.xmlreader"的 module，这就是 fullname 的作用。

在 load_next 中，我们注意到一个奇怪的现象，在 load_next 中，第二个参数是一个奇怪的 altmod，看上去它和第一个参数的作用是一样的：作为 import 执行的 package 环境。在 load_next 的代码清单 14-5 的[2]处我们看到，当 import_submodule 在环境 mod 中失败后，会转而尝试在环境 altmod 中进行 impor_submodule 的动作。这个 altmod 究竟会是什么呢？

还是考虑如图 14-25 所示的例子，在 package A 中有 module mod1，在 mod1.py 中又进行了 import sys 的动作。根据前面的结论我们知道，当 import sys 发生时，它是在 package A 中进行的，但是 package A 的环境中没有 sys module（即，该目录下没有一个名为 sys.py）的文件，sys module 是在 Python 的默认搜索路径之下的，倘若在 mod1.py 中竟然不能进行 import sys 这样的动作，那 Python 也只好找块豆腐一头撞死了。Python 提供了 altmod，正是为了解决这个问题，我们前面说了__main__ package 对应的 module 为 Py_None，而其__path__元信息就是 Python 的默认搜索路径，所以在这种情况下，altmod 为 Py_None，这使得 Python 能够在默认搜索路径下寻找在当前 package 中不存在的 module。实际上，终 Python 的一生，altmod 只能有两种选择：1. altmod == mod；2. altmod == Py_None，二者必居其一。看看前面的 import_module_level，一切就非常清晰了：

好了，了解了 import_submodule 的工作原理之后，现在我们可以进入 import_submodule 中，在这里，我们将发现之前关于 import 动作的文字描述将清晰地以代码的形式展现出来，这里，就是 import 的核心所在（见代码清单 14-6）。

**代码清单 14-6**

```
[import.c]
static PyObject* import_submodule(PyObject *mod, char *subname, char
*fullname)
{
    //获得 sys.modules
    PyObject *modules = PyImport_GetModuleDict();
    PyObject *m = NULL;

    /* 约束:
       if mod == None: subname == fullname
       else: mod.__name__ + "." + subname == fullname
    */
    //[1]: 在 sys.modules 中查找 module/package 是否已经被加载
    if ((m = PyDict_GetItemString(modules, fullname)) != NULL) {
        Py_INCREF(m);
    }
    else {
        PyObject *path, *loader = NULL;
        char buf[MAXPATHLEN+1];
        struct filedescr *fdp;
        FILE *fp = NULL;

        //[2]: 获得 import 动作的路径信息
        if (mod == Py_None)
```

```
            path = NULL;
        else {
            path = PyObject_GetAttrString(mod, "__path__");
        }

        buf[0] = '\0';
        //[3]: 搜索 module/package
        fdp = find_module(fullname, subname, path, buf, MAXPATHLEN+1, &fp,
          &loader);
        //[4]: 加载 module/package
        m = load_module(fullname, fp, buf, fdp->type, loader);
        //[5]: 将加载的 module 放入到 sys.modules 中
        add_submodule(mod, m, fullname, subname, modules);
    }
    return m;
}
```

### 14.3.2.1 搜索 module

代码清单 14-6 的[1]处首先在 Python 的 module pool 中查找是否已经加载了待加载的 module，如果是，则直接将其返回，注意，正如前面所言，这里使用了 `fullname`。

在代码清单 14-6 的[2]处，我们终于看见了 `import` 执行的 package 环境对 `import` 的影响，在这里，`__path__`元信息被抽取出来，并被传入到 `find_module` 中，作为搜索名为 `subname` 的 module 的路径信息。同样，如前面的剖析，如果 `import` 的执行环境是 __main__ package，则 `path` 将被设置为 `NULL`，在 `find_module` 中，传入的 `NULL` 的 path 将使 Python 虚拟机最终在默认的搜索路径下搜索。

Python 虚拟机在随后调用 `find_module` 的行为中，会在 path 指定的路径下搜索名为 `subname` 的 module。如果在 path 指定的路径下确实存在名为 `subname` 的 module，则有三项信息将被返回：

- `fp` 指向了被打开的实现 module 的文件；
- `buf` 中存放着 module 对应的文件在操作系统上的完整的绝对路径名；
- `fdp` 存储着关于这个 module 的元信息。

`find_module` 是一个相当复杂而繁琐的函数，在这里，我们不再深入进去，有兴趣的读者可以跟踪进入。需要说明的是，在 find_module 中，Python 将寻找一切可以作为 module 的文件，也就是说，对于 `subname`，Python 将寻找 subname.py、subname.pyc、subname.pyd、subname.pyo 和 subname.dll（Python2.5 已不支持 dll 后缀名的文件）。只要有一个文件存在，那么 `find_module` 就将成功返回。

现在问题来了，Python 如何知道发现的 module 究竟是怎样的形式呢？是 py 文件还是 C 写的 dll 文件呢？答案正是在 `find_module` 的返回值中，我们来看一看第三个返回信息

fdp 的类型 filedescr。

```
[importdl.h]
/* Definitions for dynamic loading of extension modules */
enum filetype {
    SEARCH_ERROR,
    PY_SOURCE,
    PY_COMPILED,
    C_EXTENSION,
    PY_RESOURCE, /* Mac only */
    PKG_DIRECTORY,
    C_BUILTIN,
    PY_FROZEN,
    PY_CODERESOURCE, /* Mac only */
    IMP_HOOK
};

struct filedescr {
    char *suffix;
    char *mode;
    enum filetype type;
};
extern struct filedescr * _PyImport_Filetab;
```

在 filedescr 中，保存着关于 module 的 3 个元信息：后缀名，模式及类型。Python 中维护着一个名为_PyImport_Filetab 的全局变量，它保存着所有合法的 module 的元信息。这个全局变量是在 Python 初始化时调用_PyImport_Init 构建的。在_PyImport_Init 中，Python 基于_PyImport_DynLoadFiletab 和_PyImport_StandardFiletab 两个信息来源构建_PyImport_Filetab，我们看看在 Win32 平台下，这两个信息来源是什么。

```
[import.c]
static const struct filedescr _PyImport_StandardFiletab[] = {
    {".py", "U", PY_SOURCE},
#ifdef MS_WINDOWS
    {".pyw", "U", PY_SOURCE},
#endif
    {".pyc", "rb", PY_COMPILED},
    {0, 0}

[dynload_win.c]
const struct filedescr _PyImport_DynLoadFiletab[] = {
#ifdef _DEBUG
    {"_d.pyd", "rb", C_EXTENSION},
    /* {"_d.dll", "rb", C_EXTENSION}, */
#else
    {".pyd", "rb", C_EXTENSION},
    /* {".dll", "rb", C_EXTENSION}, */
#endif
    {0, 0}
};
```

在 Python 2.5 之前，Python 是支持 dll 形式的扩展的，但是在 Python 2.5 中，sqlite3 被加入到了标准库中，同时，也引入了一些名字冲突，所以在 Python 2.5 中，_PyImport_

DynLoadFiletab 内对 dll 的支持被注释掉了，具体原因可参考 Python 源码中 dynload_win.c 的注释。

函数 find_module 返回的 fdp 正是其中的一个记录，同样，也正是这个 fdp 所携带的 module 的元信息指导了随后的 load_module 究竟应该采用怎样的方式来动态加载 module。

### 14.3.2.2 加载 module

加载 module 的动作在 load_module 中完成。

```
[import.c]
static PyObject *
load_module(char *name, FILE *fp, char *buf, int type, PyObject *loader)
{
    PyObject *modules;
    PyObject *m;
    int err;

    switch (type) {
    //py 文件
    case PY_SOURCE:
        m = load_source_module(name, buf, fp);
        break;
    //pyc 文件
    case PY_COMPILED:
        m = load_compiled_module(name, buf, fp);
        break;
    //dll(pyd)文件
    case C_EXTENSION:
        m = _PyImport_LoadDynamicModule(name, buf, fp);
        break;
    //加载 package
    case PKG_DIRECTORY:
        m = load_package(name, buf);
        break;
    case C_BUILTIN:
        if (buf != NULL && buf[0] != '\0')
            name = buf;
        //创建内建 module
        init_builtin(name);
        //确认内建 module 出现在 sys.modules 中，如果没有，则抛出异常
        modules = PyImport_GetModuleDict();
        m = PyDict_GetItemString(modules, name);
        if (m == NULL) {
            ......//抛出异常
            return NULL;
        }
        Py_INCREF(m);
        break;
        ......
    }
```

```
    return m;
}
```

Python 虚拟机会根据 module 的不同类型选择不同的加载方式。

### py 文件与 pyc 文件

对于 py 文件，Python 虚拟机会执行 load_source_module，我们不再追入进去，但是我们可以猜想，在 load_source_module 中，Python 虚拟机一定先对 py 文件进行了编译，产生了 PyCodeObject 对象，然后执行了 PyCodeObject 对象中的字节码，因为唯有如此，才能通过执行 def、class 等语句创建 PyFunctionObject、PyClassObject 等对象，才能在最后得到一个从符号映射到对象的 dict。这个 dict 自然也就是在 load_source_module 中所创建的 module 对象中维护的那个 dict。既然说 load_source_modules 中创建了 module，那么可以猜想，在 load_source_modules 中，创建的 module 已经被放置到了全局 module 集合 sys.modules 中了。

对于 pyc 文件，Python 虚拟机将调用 load_compiled_module，这个函数的动作与 load_source_module 其实是非常相似的，只是少了编译的动作。

### package

对于 package，Python 虚拟机会调用 load_package（见代码清单 14-7）。

代码清单 14-7

```
[import.c]
static PyObject *load_package(char *name, char *pathname)
{
    PyObject *m, *d;
    PyObject *file = NULL;
    PyObject *path = NULL;
    int err;
    char buf[MAXPATHLEN+1];
    FILE *fp = NULL;
    struct filedescr *fdp;

    //[1]: 创建 PyModuleObject 对象，并加入 sys.modules 中
    m = PyImport_AddModule(name);
    d = PyModule_GetDict(m);
    ......
    //[2]: 在 package 的目录下寻找并加载__init__.py 文件
    fdp = find_module(name, "__init__", path, buf, sizeof(buf), &fp, NULL);
    m = load_module(name, fp, buf, fdp->type, NULL);
    ......
    return m;
}
```

我们之前说了 package 同样也是一种 module，在 load_package 的代码清单 14-7 的[1]处，这一点显露无疑。同时在代码清单 14-7 的[2]处，我们也看到了为什么一个目录下必

须有__init__.py 才能被 Python 认为是一个合法的 package。虽然这里的代码没有列出，但是你可以想象，一旦 find_module 失败，Python 必定会抛出异常。

值得注意的是，load_package 只会将 package 自身加载到 Python 中，并不会对 package 中的 module 进行任何动作，除非在__init__.py 中有显式的 import 语句。在 load_package 中，find_module 和 load_module 所搜索的和加载的都仅仅是__init__.py，如果__init__.py 为空，那么 load_package 就不会加载 package 下的任何 module，而是沿着函数调用栈一层一层向上返回到 import_module_level 中，通过下一个 load_next 来加载 package 中的 module。

### 内建 module

对于 Python 内建的 module，有一部分在 Python 初始化运行环境时已经被加载到了 sys.modules 中，而还有一部分并不是很常用的则没有被加载到 sys.modules 中，比如说 math module。假如用户在运行时进行 import math 的动作，那么 Python 虚拟机就会通过 init_builtin("math")来加载 math module（见代码清单 14-8）。

**代码清单 14-8**

```
[import.c]
static int init_builtin(char *name)
{
    struct _inittab *p;
    //[1]: 在内建 module 的备份中查找名为 name 的 module
    if (_PyImport_FindExtension(name, name) != NULL)
        return 1;
    //[2]: 遍历内建 module 集合，寻找匹配的 module
    for (p = PyImport_Inittab; p->name != NULL; p++) {
        if (strcmp(name, p->name) == 0) {
            //[3]: 初始化内建 module
            (*p->initfunc)();
            //[4]: 加入到内建 module 备份中
            _PyImport_FixupExtension(name, name);
            return 1;
        }
    }
    return 0;
}
```

还记得我们在上一章剖析 Python 初始化过程时提到的吗？Python 在内部维护了一个对于内建 module 的备份，所有的内建 module，一旦被加载之后，都会拷贝一份，存到这个备份中，以后再次 import 时就可以不用再进行 module 的初始化动作，直接使用备份中缓存的 module 就可以了。

在 init_builtin 中，第一步正是在代码清单 14-8 的[1]处搜索这个内建 module 的备份，如果搜索到了，则直接返回。如果内建 module 备份中并没有存储我们需要的 module，

那么 Python 虚拟机只好自己创建 module，并进行初始化。

在 Python 中，PyImport_Inittab 是一个全局变量，这个变量维护着一个内建 module 的完整列表，代码清单 14-8 的[2]处的动作就是遍历这个列表，在列表中查找我们期望的内建 module。

```
[import.h]
struct _inittab {
    char *name;
    void (*initfunc)(void);
};

[import.c]
struct _inittab *PyImport_Inittab = _PyImport_Inittab;

[PC\config.c]
struct _inittab _PyImport_Inittab[] = {
        ......
        {"math", initmath},
        {"nt", initnt}, /* Use the NT os functions, not posix */
        ......
};
```

我们所期望的 math module 正是在这个内建 module 列表中。Python 虚拟机在找到之后，会在 init_builtin 代码清单 14-8 的[3]处通过调用 initmath 来创建并初始化 math module。

```
[modsupport.h]
#define Py_InitModule3(name, methods, doc) \
    Py_InitModule4(name, methods, doc, (PyObject *)NULL,PYTHON_API_VERSION)

[mathmodule.c]
void initmath(void)
{
    PyObject *m, *d, *v;

    m = Py_InitModule3("math", math_methods, module_doc);
    ......
}
```

这里我们看到了已经非常熟悉的 Py_InitModule4，没错，正是在 Py_InitModule4 中，Python 虚拟机创建了 PyModuleObject 对象，并将其加入到了 sys.modules 中，并且，还将 math_methods 中包含的每一个 C 函数指针都包装成了一个 PyCFunctionObject 对象，加入到了所创建的 PyModuleObject 对象的名字空间之中。

到此，对内建对象的加载动作也就大功告成了。

### C 扩展 module

Python 的一大卖点就是可以和 C/C++等无缝集成，但是受 Python 的设计所限，并不是随意编写的一个 C/C++的模块都能被 Python 所接受，而是必须按照一定的规则来编写

Python 的扩展模块。Python 是如何调用 C 语言编写的扩展 module？对这一点的剖析关系到能否深刻理解为什么要以这样的规则来编写 Python 扩展模块。

在 load_module 中，如果 import 动作的目标是 C 语言编写的 Python 扩展 module，那么 Python 将调用_PyImport_LoadDynamicModule。

**代码清单 14-9**

```
[importdl.c]
PyObject *_PyImport_LoadDynamicModule(char *name, char *pathname, FILE *fp)
{
    PyObject *m;
    char *lastdot, *shortname, *packagecontext, *oldcontext;
    dl_funcptr p;
    //[1]: 在 Python 的 module 备份中检查是否有名为 name 的 module
    if ((m = _PyImport_FindExtension(name, pathname)) != NULL) {
        Py_INCREF(m);
        return m;
    }
    ……
    //[2]: 从 dll 中获得 module 的初始化函数的起始地址
    p = _PyImport_GetDynLoadFunc(name, shortname, pathname, fp);
    //[3]: 调用 module 的初始化函数
     (*p)();
    //[4]: 从 sys.modules 中获得已经被加载的 module
    m = PyDict_GetItemString(PyImport_GetModuleDict(), name);
    //[5]: 设置 module 的__file__属性
    PyModule_AddStringConstant(m, "__file__", pathname);
    //[6]: 将 module 加入到 Python 的 module 备份中
    _PyImport_FixupExtension(name, pathname);
    return m;
}
```

需要指出的是，在不同的操作系统中，用 C 编写 Python 扩展 module 得到的结果是不同的，比如在 Windows 下结果是 dll（pyd），而在 Linux 下就是 so。所以加载 Python 扩展 module 的函数是和平台相关的，这里我们剖析的是加载 windows 下 dll 文件中 Python 扩展 module 的代码，其他平台的加载动作与此类似。

可以看到，Python 对于扩展 module，也动用了 module 备份，在代码清单 14-9 的[1]处，如果备份中有此 module，那么 Python 虚拟机将直接返回；如果备份中没有此 module，则 Python 虚拟机会加载该 module，并在代码清单 14-9 的[5]处将加载后的 module 拷贝一份到 module 备份中。

代码清单 14-9 的[4]、[5]两处只是为了在已经加载的 module 中添加一个__file__属性。在代码清单 14-9 的[4]之前，Python 虚拟机已经完成了 module 的加载工作。所以，对于加载 dll 中的 Python 扩展 module，关键的代码在代码清单 14-9 的[2]和[3]处。

代码清单 14-9 的[2]会打开拥有 Python 扩展 module 的 dll 文件，并从该 dll 文件中获

得 module 初始化函数的起始地址。在进入[2]处的_PyImport_GetDynLoadFunc 之前，我们先来看一看这时两个非常重要的参数。假如我们进行了“import abc”的动作，而 abc module 是在 abc.dll 中实现的，那么在代码清单 14-9 的[2]处，name 的值为“abc”，而 shortname 的值也为“abc”（见代码清单 14-10）。

代码清单 14-10

```
[dynload_win.c]
dl_funcptr
_PyImport_GetDynLoadFunc(char *fqname, char *shortname, char *pathname,
FILE *fp)
{
    dl_funcptr p;
    char funcname[258], *import_python;
    //[1]: 获得module的初始化函数名
    PyOS_snprintf(funcname, sizeof(funcname), "init%.200s", shortname);
    {
        HINSTANCE hDLL = NULL;
        //[2]: 使用Win32 API加载dll文件
        hDLL = LoadLibraryEx(pathname, NULL,
        LOAD_WITH_ALTERED_SEARCH_PATH);
        if (hDLL==NULL){
            …… //加载dll文件失败，抛出异常
        } else {
            char buffer[256];
            //[3]: 获得当前Python对应的dll文件名
#ifdef _DEBUG
            PyOS_snprintf(buffer, sizeof(buffer), "python%d%d_d.dll",
#else
            PyOS_snprintf(buffer, sizeof(buffer), "python%d%d.dll",
#endif
                    PY_MAJOR_VERSION,PY_MINOR_VERSION);
            //[4]: 获得module中所引用的Python的dll文件名
            import_python = GetPythonImport(hDLL);
            //[5]: 确保当前Python对应的dll即是module所引用的dll
            if (import_python && strcasecmp(buffer,import_python)) {
                ……//dll文件不匹配，抛出异常
            }
        }
        //[6]: 调用Win32 API获得module初始化函数的地址
        p = GetProcAddress(hDLL, funcname);
    }
    return p;
}
```

在代码清单 14-10 的[1]处，Python 虚拟机获得了 module 中初始化函数的函数名，在这里，abc module 的初始化函数名为 initabc。到了这里，我们应该能够理解为什么在用 C 编写名为 usermodule 的 Python 扩展 module 时，一定要有一个 initusermodule 函数了。这其实是一个协议，只有在我们编写的扩展 module 遵循 Python 设计时所采用的这个协议，Python 才能和扩展 module 建立起联系。

Python 虚拟机在代码清单 14-10 的[2]处调用了 Win32 API，将 dll 文件加载到内存中。如果加载失败，那么自不必言，虚拟机将抛出异常。然而奇怪的是，在加载成功之后，Python 虚拟机并不急于获得 module 初始化函数的地址，而是在代码清单 14-10 的[3]、[4]、[5]处进行了一连串的奇怪动作。这些动作有什么意义呢？

在前面分析 Python 虚拟机的开始，我们就曾看到，在 Python 的发展历史上，Python 所使用的字节码指令在不断发生改变。同样，Python 向外所暴露的 C 接口也可能发生改变。假如我们的扩展 module 在创建时链接的是 python25.dll，而我们当前使用的 Python 环境是 Python 2.0（python20.dll），这就隐藏着一个可能，即扩展 module 中所使用的 C 接口并没有在 python20.dll 中提供。Python 为了避免这样的不兼容情况的出现，在代码清单 14-10 的[3]、[4]、[5]处进行了兼容性的检查。Python 虚拟机会检查当前运行的 Python 所使用的 dll 文件是否与扩展 module 所引用的 dll 文件匹配，如果不匹配，则 Python 虚拟机会抛出异常。

如果加载 dll 文件顺利，而且也通过了兼容性检查，那么 Python 虚拟机将再次调用 Win32 API，获得 module 初始化函数的地址。

这样，程序的流程回到了`_PyImport_LoadDynamicModule`的代码清单 14-9 的[3]处，在这里，Python 虚拟机开始调用 module 的初始化函数(`initabc`)，完成了 module 的加载工作。需要特别注意的是，这个初始化函数是在 dll 中提供的，我们来看一个用 C 编写 Python 扩展 module 的例子。

```
[C extension module : abc.dll]
static PyMethodDef abc_methods[] = {
    {"hello", Hello, METH_VARARGS, "say hello"},
    {NULL, NULL}
};

EXPORT int initabc(void)
{
    Py_InitModule("abc", abc_methods);
    return 0;
}
```

注意：在`initabc`中，又调用了 Python 的 C 接口。回想一下前面我们刚刚剖析过的 Python 虚拟机对内建 module 的加载动作，那里出现了一个“`Py_InitModule3( "math", math_methods, math_doc)`”，而这里出现了一个“`Py_InitModule( "abc", abc_methods)`”，简直太像了，现在我们敢打赌，C 扩展 module 的加载实质与内建 module 的加载是一模一样的。我们来看看`Py_InitModule`。

```
[modsuport.h]
#define Py_InitModule(name, methods) \
    Py_InitModule4(name, methods, (char *)NULL, (PyObject *)NULL, \
                PYTHON_API_VERSION)
```

没错，原来 Py_InitModule 和 Py_InitModule3 是一样的，都只是 Py_InitModule4 的化身而已。对于 Py_InitModule4，我们已经很熟悉了。所以现在可以清楚地知道，Python 将在 sys.modules 中创建一个名为 abc 的 module。然后遍历 abc_methods 中的每一个 PyMethodDef 结构体，并创建 PyCFunctionObject 对象，将此对象和其对应的操作名作为属性添加到 abc module 中。正如我们所料，与加载内建 module 的过程一模一样。

### 14.3.3 from 与 import

在第 3 节的开始，我们就已经提到，当 Python 虚拟机执行的是 “from a import b, c” 时，import_module_level 中的 fromlist 参数将不为 Py_None，这将导致 Python 虚拟机进行一些额外的动作，同时会导致 import_module_level 的返回值发生改变（见代码清单 14-11）。

**代码清单 14-11**

```
[import.c]
static PyObject *
import_module_level(char *name, PyObject *globals, PyObject *locals,
PyObject *fromlist, int level)
{
    ……
    //[3]: 处理 from *** import *** 语句
    if (fromlist != NULL) {
        if (fromlist == Py_None || !PyObject_IsTrue(fromlist))
            fromlist = NULL;
    }
    //import 的形式不是 from *** import ***，返回 head
    if (fromlist == NULL) {
        return head;
    }

    //import 的形式是 from *** import ***，返回 tail
    if (!ensure_fromlist(tail, fromlist, buf, buflen, 0)) {
        return NULL;
    }
    return tail;
}
```

当 fromlist 不为空时，Python 虚拟机将进行一个 ensure_fromlist 的动作，这个动作到底做了些什么事呢？考虑一下 “from a import b, c”，如果这个语句能成功完成，那么一定意味着，在符号 “a” 对应的 module 对象的名字空间中，必有两个符号，一个名为 “b”，另一个名为 “c”。

更进一步，“b” 或 “c” 这两个符号可以不在 “a” 对应的 module 对象的名字空间中，但通过 “a”，必须能够发现符号 “b” 或 “c”，比如在图 14-25 所示的例子中，如果执行

"from A import mod2"，尽管 mod2 并不在 A 对应的 module 对象的名字空间中，但是 import 机制能够根据"A"发现"mod2"，所以，这也是合法的。

函数 ensure_fromlist 的作用就是确保在 tail 中能够发现 fromlist 中的所有符号，这时的 tail 对应的是一个 module 对象。

```
[import.c]
static int ensure_fromlist(PyObject *mod, PyObject *fromlist, char *buf,
                           Py_ssize_t buflen, int recursive)
{
    int i;
    for (i = 0; ; i++) {
        PyObject *item = PySequence_GetItem(fromlist, i);
        int hasit;
        //若 item 为 NULL，则结束 ensure 动作
        if (item == NULL) {
            return 0;
        }

        if (PyString_AS_STRING(item)[0] == '*') {
            PyObject *all;
            /* See if the package defines __all__ */
            all = PyObject_GetAttrString(mod, "__all__");
            int ret = ensure_fromlist(mod, all, buf, buflen, 1);
            continue;
        }
        hasit = PyObject_HasAttr(mod, item);
        //hasit 为 false，意味着出现"from A import mod2"这样的情形
        if (!hasit) {
            char *subname = PyString_AS_STRING(item);
            PyObject *submod;
            char *p;
            p = buf + buflen;
            *p++ = '.';
            strcpy(p, subname);
            submod = import_submodule(mod, subname, buf);
        }
    }
}
```

到了这里，你能够理解 py 文件中通常出现的__all__到底是何方神圣了吧☺。

## 14.4 Python 中的 import 操作

在本节中，我们将研究 Python 中各种 import 操作所对应的字节码指令序列。通过本节的剖析，我们就能更加深入地了解 Python 中各种 import 操作所完成的动作。

## 14.4.1 import module

```
import sys
#  LOAD_CONST      0 (-1)
#  LOAD_CONST      1 (None)
#  IMPORT_NAME     0 (sys)
#  STORE_NAME      0 (sys)
```

这是我们在本章开头考察的例子，现在我们已经清楚地了解了 IMPORT_NAME 的行为，在 IMPORT_NAME 指令的最后，Python 虚拟机会通过 SET_TOP 将加载后的 PyModuleObject 对象压入到运行时栈内，所以随后的 STORE_NAME 指令会将（“sys”, module object）元素对存放到当前的 local 名字空间中。

## 14.4.2 import package

```
import xml.sax.xmlreader
#  LOAD_CONST      0 (-1)
#  LOAD_CONST      1 (None)
#  IMPORT_NAME     0 (xml.sax.xmlreader)
#  STORE_NAME      1 (xml)
```

如果在 import 的动作中涉及到对 package 的 import 动作，那么 IMPORT_NAME 指令的指令参数将是关于 module 的完整的“路径”信息，IMPORT_NAME 指令的内部将解析这个“路径”，并为“xml”,“xml.sax”和“xml.sax.xmlreader”都创建一个 PyModuleObject 对象，最后将与“xml”对应的 PyModuleObject 对象返回（参考 imort_module_level 的代码，这种情况下返回的是 head，正好对应“xml”）。所以在 IMPORT_NAME 之后的 STORE_NAME 指令只是在当前的 local 名字空间中加入了一个“xml”符号。

## 14.4.3 from & import

```
from xml.sax import xmlreader
#  LOAD_CONST      0 (-1)
#  LOAD_CONST      1 (('xmlreader', ))
#  IMPORT_NAME     0 (xml.sax)
#  IMPORT_FROM     1 (xmlreader)
#  STORE_NAME      1 (xmlreader)
```

注意：这里的“LOAD_CONST 1”指令的结果不再是 None 了，而是一个 tuple 对象，也就是我们之前在 14.3.3 节中提到的 fromlist。

随后的“IMPORT_NAME 0”最终在 import_module_level 中将返回 tail，也就是符号“xml.sax”对应的 module 对象（这一点非常重要，理解了这一点，你就能理解 Python 内建函数__import__在函数参数 fromlist 不同时的不同行为了），以供指令 IMPORT_FROM

使用。

```
[IMPORT_FROM]
            w = GETITEM(names, oparg);
            v = TOP();
            x = import_from(v, w);
            PUSH(x);
```

现在已经很清楚了，在 IMPORT_FROM 指令执行时，w 是值为"xmlreader"的 PyStringObject 对象，而 v 则是在 IMPORT_NAME 指令中返回的那个 tail，那么 IMPORT_FROM 指令代码的关键就在于那个 import_from 函数了。

```
[ceval.c]
static PyObject *import_from(PyObject *v, PyObject *name)
{
    PyObject *x;

    x = PyObject_GetAttr(v, name);
    if (x == NULL && PyErr_ExceptionMatches(PyExc_AttributeError)) {
        PyErr_Format(PyExc_ImportError,
                  "cannot import name %.230s",
                  PyString_AsString(name));
    }
    return x;
}
```

非常简单，其实就是在"xml.sax"对应的 module 对象的名字空间中搜索符号"xmlreader"，显然，这个搜索动作是能够成功的。

最后会由 STORE_NAME 指令将符号"xmlreader"与 import_from 中搜索得到的结果捏在一起，存放到当前的 local 名字空间中。

所以，最后在 local 名字空间中，只出现了我们期望的"xmlreader"符号，尽管 xml.sax 所对应的 module 对象被创建了，并且也被加入到了 sys.modules 中，但这个过程被 Python 隐藏了，如果仅仅从操作的结果来看的话，我们对这些中间的动作也是一无所知的。

对于 from 和 import 的组合，还有一种"from xml import *"这样的形式，这种形式牵涉到在 module 文件中的一个特殊符号："__all__"，我们利用这个符号可以控制 module 中想要暴露给外界的符号。"from xml import *"编译的结果中最关键的是 IMPORT_STAR 这个指令，而指令的实现代码中最重要的部分则最终归结到 import_from_all 这个函数，这里就不深入了。有兴趣的读者自行剖析一下 import_from_all 函数，看看__all__这个符号是如何影响 module 的载入的。

### 14.4.4 import & as

```
from xml.sax import xmlreader as myreader
#  LOAD_CONST        0 (-1)
```

```
#  LOAD_CONST        1 (('xmlreader', ))
#  IMPORT_NAME       0 (xml.sax)
#  IMPORT_FROM       1 (xmlreader)
#  STORE_NAME        2 (myreader)
```

在 import 动作中加入符号重命名的机制非常简单，动态加载的过程与我们之前的分析完全一致，只是在通过 STORE_NAME 指令向当前的 lcoal 名字空间引入符号时，加入的是我们通过“as”指定的符号“myreader”，而不再是“xmlreader”了。

## 14.4.5 reload

```
import stemer
#  LOAD_CONST        0 (-1)
#  LOAD_CONST        1 (None)
#  IMPORT_NAME       0 (stemer)
#  STORE_NAME        0 (stemer)
reload(stemer)
#  LOAD_NAME         1 (reload)
#  LOAD_NAME         0 (stemer)
#  CALL_FUNCTION     1
```

Python 虚拟机最终将调用 builtin module 中的 reload 操作（对应函数 builtin_reload）来完成重新载入的动作。在 builtin_reload 中，将最终调用 load_module，喏，就是这个我们之前详细分析过的家伙。

假如 stemer module 是由 py 文件实现的 module，那么在 load_module 中，Python 虚拟机将调用与 py 文件对应的加载操作 load_source_module。而在 load_source_module 这个函数中，将进入 PyImport_ExecCodeModuleEx，在这个函数中，我们就能看到为什么在 stemer.py 中的删除操作不会影响到 sys.modules 中的那个与 stemer module 对应的 module 对象（见代码清单 14-12）。

**代码清单 14-12**

```
[import.c]
PyObject *PyImport_ExecCodeModuleEx(char *name, PyObject *co, char
  *pathname)
{
    //[1]: 获得 sys.modules
    PyObject *modules = PyImport_GetModuleDict();
    PyObject *m, *d, *v;
    //[2]: 从 sys.modules 中获得 name 对应的 module 对象 (name = "stemer")
    //     该 module 在 sys.modules 中已经存在
    m = PyImport_AddModule(name);
    /* If the module is being reloaded, we get the old module back
       and re-use its dict to exec the new code. */
    //[3] : stemer module 已经存在，使用其 module 对象中的 dict
    d = PyModule_GetDict(m);
    //[4] : 以 d 作为 local 名字空间和 global 名字空间，最终调用 PyEval_EvalFrame
```

```
    //      向 d 中更新或添加新的符号
    v = PyEval_EvalCode((PyCodeObject *)co, d, d);
    ......
}
```

在代码清单 14-12 的[4]处，只是向 stemer.py 对应的 module 对象中更新或添加了新的符号。如果有符号在源文件中被删除，那么这种删除是不会影响到 stemer 的 module 对象中的 dict 的。

### 14.4.6 内建 module：imp

在 py 源代码文件中的 import 语句是静态的，其实，Python 也提供了动态的 import 机制，即在运行时动态地选择需要 import 的对象，Python 通过一个 imp 的内建 module 暴露了我们之前所考察的用于 import 机制的核心接口。

我们说 imp 是一个内建 module，那么根据我们现在的经验，一定有一个 imp_methods 与之对应，没错，这个 imp_methods 正是在 import.c 中。

```
[import.c]
static PyMethodDef imp_methods[] = {
    {"find_module",  imp_find_module,  METH_VARARGS, doc_find_module},
    ......
    {"load_module",  imp_load_module,  METH_VARARGS, doc_load_module},
    ......
    {"load_source",     imp_load_source,     METH_VARARGS},
    {NULL,          NULL}       /* sentinel */
};
```

这里，我们就不再详细地剖析 imp_methods 了，有兴趣的读者可以参考 Python 的源代码，你会发现，这些操作其实就是对我们之前讨论的 Python 中 import 机制的核心函数的包装而已。

## 14.5 与 module 有关的名字空间问题

在“Python 虚拟机框架”一章中，当我们初次系统性地接触名字空间时，其中有一个例子是与 module 相关的，来回顾一下（见代码清单 14-13）。

**代码清单 14-13**

```
[module1.py]
import module2

owner = 'module1'
module2.show_owner() //[1]

[module2.py]
```

```
owner = 'module2'

def show_owner():
    print owner //[2]
```

尽管在 module1.py 中，符号“owner”对应的是<string 'module1'>，但是在代码清单 14-13 的[1]处，输出的结果却是 module2.py 中的符号“owner”对应的值。我们说这是 Python 的 LEGB 规则所导致的结果，因为 print owner 这个语句是在 module2.py 中，所以根据 LEGB 规则，找到的是在 module2.py 中的符号 owner。但是，问题是，这一切是怎么实现的？

根据上一章中对初始化时名字空间的分析，我们知道，在代码清单 14-13 的[1]之前，global 名字空间中一定有一个符号“owner”，且符号关联着<string 'module1'>。好，我们接受 LEGB，但在调用 show_owner 时，不是应该在 global 名字空间中找到<string 'module1'>才对吗？看来唯一的答案就是其中这个 global 名字空间发生了变化，问题的关键在于函数调用机制。

为了搞清楚这一切，我们先来看看 module2.show_owner()这条表达式的编译结果。

```
[module1.py]
import module2

owner = 'module1'
module2.show_owner()
18 LOAD_NAME     0 (module2)
21 LOAD_ATTR     2 (show_owner)
24 CALL_FUNCTION  0
```

Python 虚拟机执行“18 LOAD_NAME 0”的结果是得到了一个 PyModuleObject 对象，而“21 LOAD_ATTR 2”指令则是一个纯粹的名字空间搜索指令，它将使 Python 虚拟机在 PyModuleObject 对象的名字空间中搜索符号“show_owner”，这个符号显然对应着一个 Python 虚拟机在对 module2.py 进行 import 动作时创建的 PyFunctionObject 对象。图 14-26 展示了这一过程。

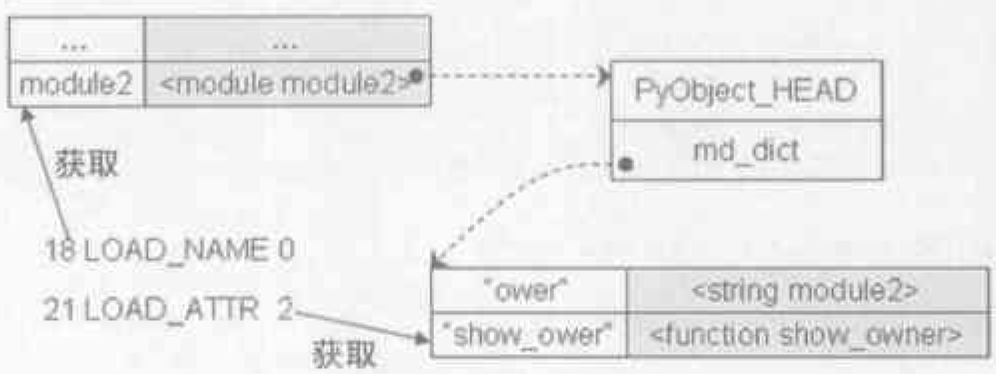



图 14-26 Python 虚拟机获得符号“show_ower”的过程

但是，在这两条指令中，都不会改动 global 名字空间，而下一步就是“24 CALL_FUNCTION 0”这条函数调用指令了，而改动 global 名字空间的动作就发生在这里。

回忆一下前面剖析的函数调用机制，我们知道，函数调用实际上会创建一个新的“栈帧”（PyFrameObject 对象），在这个 PyFrameObject 对象的环境中执行函数内的指令。再回忆下创建 PyFrameObject 对象的函数，我们会发现，果然就有一个设置 global 名字空间的动作。

```
[frameobject.c]
PyFrameObject *
PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject *globals,
        PyObject *locals)
{
    ......
    //设置 global 名字空间
    f->f_globals = globals;
    ......
    return f;
}
```

但是俗话说得好，一个巴掌拍不响，光有 PyFrame_New 内部妄图改朝换代的愿望还不够，还得 Python 虚拟机在调用 PyFrame_New 时传入一个不同于 module1.py 中 global 名字空间的 dict 对象。这个对象恰恰就在图 14-26 中符号“show_owner”对应的 PyFunctionObject 对象中。

Python 对 module2.py 的编译结果中，“def show_ower():”这条表达式会对应一条 MAKE_FUNCTION 指令。来回顾一下这条指令都干了什么。

```
typedef struct {
    PyObject_HEAD
    ......
    PyObject *func_globals; /* A dictionary (other mappings won't do) */
    ......
} PyFunctionObject;

[MAKE_FUNCTION 的指令代码]
        ......
        x = PyFunction_New(v, f->f_globals);
        ......

PyObject * PyFunction_New(PyObject *code, PyObject *globals)
{
    PyFunctionObject *op =
      PyObject_GC_New(PyFunctionObject,&PyFunction_Type);
    ......
    op->func_globals = globals;
    ......
    return (PyObject *)op;
}
```

当 Python 虚拟机在 module1.py 中碰到“import module2.py”时，会对 module2.py 进行 import 动作，实际上，也就是执行以下 module2.py，在这个过程中，MAKE_FUNCTION 指令就会被 Python 虚拟机执行，我们看到，确实，MAKE_FUNCTION 将当前活动的

PyFrameObject 对象的 global 名字空间保存到了它所创建的 PyFunctionObject 对象中。

在本章 14.3.2.2 节中，我们看到，对.py 文件的 import 动作最终将委托给 load_source_module 函数，如果你顺着这个函数追下去，会最终发现对 PyEval_EvalCodeEx 的调用，这意味着 Python 在 import 时，会创建新的 PyFrameObject 对象，这个 PyFrameObject 对象中的 global 名字空间最初会是一个空的 dict 对象，但是关键在于，这个新建的 PyFrameObject 对象中的 local 名字空间和 global 空间都指向同一个 dict 对象（回忆一下上一章对 Python 初始化时第一个名字空间创建过程的观察），所以在执行 module2.py 中的代码时，会影响 local 名字空间，也就影响了 global 名字空间，最终 MAKE_FUNCTION 指令创建的 PyFunctionObject 对象中的 func_globals 中也就包含了 module2.py 中的所有符号。

现在只剩下最后一步了，在 module1.py 中，对 show_owner 函数进行调用时，在某个地方，Python 虚拟机一定抽取了 show_owner 对应的 PyFunctionObject 对象中的包含 module2.py 中所有符号的 func_globals，并将其传递给了 PyFrame_New，这样才能保证 LEGB 规则最终找到的是 module2.py 中的符号"ower"。好了，fast_function，该你出场了。

```
[ceval.c]
#define PyFunction_GET_GLOBALS(func)  (((PyFunctionObject *)func) ->
 func_globals)

static PyObject *
fast_function(PyObject *func, PyObject ***pp_stack, int n, int na, int nk)
{
    PyObject* f
    PyObject *globals = PyFunction_GET_GLOBALS(func);
    ......
    return PyEval_EvalCodeEx(co, globals,
                (PyObject *)NULL, (*pp_stack)-n, na,
                (*pp_stack)-2*nk, nk, d, nd,
                PyFunction_GET_CLOSURE(func));
}

PyObject *
PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
      PyObject **args, int argcount, PyObject **kws, int kwcount,
      PyObject **defs, int defcount, PyObject *closure)
{
......
f = PyFrame_New(tstate, co, globals, locals);
......
}
```

正是在 fast_function 中，一个全新的 global 名字空间产生了，并最终维护了 LEGB 规则的正确性。这么一个简单的现象，却涉及了 Python 内部运作的方方面面，但最关键

的一点，还是名字空间，名字空间可谓是 Python 的灵魂所在。当然，更准确地说，还有另外一个灵魂：最内嵌套作用域规则，正是这个规则决定了这些实现。

CHAPTER

第 15 章

# Python 多线程机制

开发多线程的应用系统，是在日常的软件开发中经常会遇到的需求。现在的编程语言都为多线程开发提供了很好的支持，无论是通过库的支持还是将多线程机制内建在语言之中。Python 也为多线程系统的开发提供了很好的支持。

同样身为动态语言，Ruby 也提供了多线程的支持，但是在 Ruby 1.9 之前的多线程机制是在语言的实现中模拟了线程及线程调度机制，而并没有使用操作系统本身的线程机制（在以后的描述中，我们称为原生线程）。Ruby 1.9 中整合了 YARV 作为 Ruby 新的虚拟机，在 YARV 中，将操作系统的原生线程引入了 Ruby。每一个 Ruby 线程都对是操作系统上的一个线程，在 Ruby 内部，维护着一个全局资源锁，一个 Ruby 线程必须首先获得这个锁，才能成为活动的线程，从而使用 Ruby 虚拟机的全局资源。

这一切，在 Python 中早已实现，Python 中的线程从一开始就是操作系统的原生线程，而 Python 虚拟机也同样使用一个全局解释器锁（Global Interpreter Lock，GIL）来互斥线程对 Python 虚拟机的使用。

## 15.1 GIL 与线程调度

为了理解 Python 为什么需要 Global Interpreter Lock（GIL），考虑这样的情形：假设有两个线程 A、B，在两个线程中，都同时保存着对内存中同一对象 obj 的引用，也就是说，这时 obj->ob_refcnt 的值为 2。如果 A 销毁对 obj 的引用，显然，A 将通过 Py_DECREF 调整 obj 的引用计数值。我们知道，Py_DECREF 的整个动作可以分为两个部分：

- `--obj->ob_refcnt;`
- `if(obj->ob_refcnt == 0) destory object and free memory。`

如果 A 在执行完第一个动作之后，obj->ob_refcnt 的值变为 1。不幸的是，恰恰在

这个时候，线程调度机制将 A 挂起，而唤醒了 B。更为不幸的是，B 同样也开始销毁对 obj 的引用。B 完成第一个动作之后，obj->ob_refcnt 为 0，B 是一个幸运儿，它没有被线程调度打断，而是顺利地完成了接下来的第二个动作，将对象销毁，内存释放。好了，现在 A 又被重新唤醒，可惜现在已经物是人非，obj->ob_refcnt 已经被 B 减少到 0，而不是当初的 1。按照约定，傻乎乎的 A 开始再一次地对已经销毁的对象进行对象销毁和内存释放的动作。这样的结局是什么？只有天知道。

为了支持多线程机制，一个基本的要求就是需要实现不同线程对共享资源访问的互斥，Python 也不例外，这正是引入 GIL 的根源所在。Python 中的 GIL 是一个非常霸道的互斥实现，正如它的名字所暗示的，GIL 是一个解释器（Interpreter）——为了呼应 GIL 中的 Interpreter，本章中，我们会以解释器来称呼虚拟机——级的互斥机制，也就是说，在一个线程拥有了解释器的访问权之后，其他的所有线程都必须等待它释放解释器的访问权，即使这些线程的下一条指令并不会互相影响。初看上去，这样的保护机制粒度太大了，我们似乎只需要将可能被多个线程共享的资源保护起来即可，对于不会被多个线程共享的资源，完全可以不用保护。实际上，在 Python 的发展历史中，确实出现过这样的解决方案，但是令人惊奇的，这样的方案在单处理器上的多线程实现的效率上却没有 GIL 的方案好。所以现在 Python 中的多线程机制是在 GIL 的基础上实现的。

当然，这样的方案也意味着，无论如何，在同一时间，只能有一个线程能访问 Python 所提供的 API。注意这里的同一时间对于单处理器是毫无意义的，因为单处理器的本质是不可能并行的，但是对于多处理器，情形就完全不同了，同一时间，确实可以有多个线程独立运行，然而 Python 的 GIL 限制了这样的情形，使得多处理器最终退化为单处理器，性能大打折扣。这一点其实早已被 Python 社区所认识，也进行了大量的探索。大约在 99 年的时候，Greg Stein 和 Mark Hammond 两位老兄基于 Python 1.5 创建了一份去除 GIL 的 branch，但是很不幸，这个分支在很多基准测试上，尤其是单线程操作的测试上，效率只有使用 GIL 的 Python 的一半左右。因为细粒度的锁机制会导致大量的加锁、解锁的操作，而加锁、解锁对于操作系统来说，是一个比较重量级的动作；另一方面，没有了 GIL 的保护，编写 Python 扩展模块的难度大大增加。所以，到目前 Python 的最新版本 2.5 为止，GIL 仍然是多线程机制的基石，而我们也仍然将视线集中在单处理器上。实际上，在去年 5 月份 Python 3000 的邮件列表上，Python 的创造者，Guido，提出了一个比较可行的的解决方案，在多处理器的情况下，完全可以创建多个 Python 进程，充分使用多处理器，进程之间通过 IPC 的方式进行通信。当然，Guido 也仅仅是提出了这么一个想法，并没有太多的实现细节透露出来。

图 15-1 显示了我们对 Python 的多线程机制所建立的一个粗略的模型。

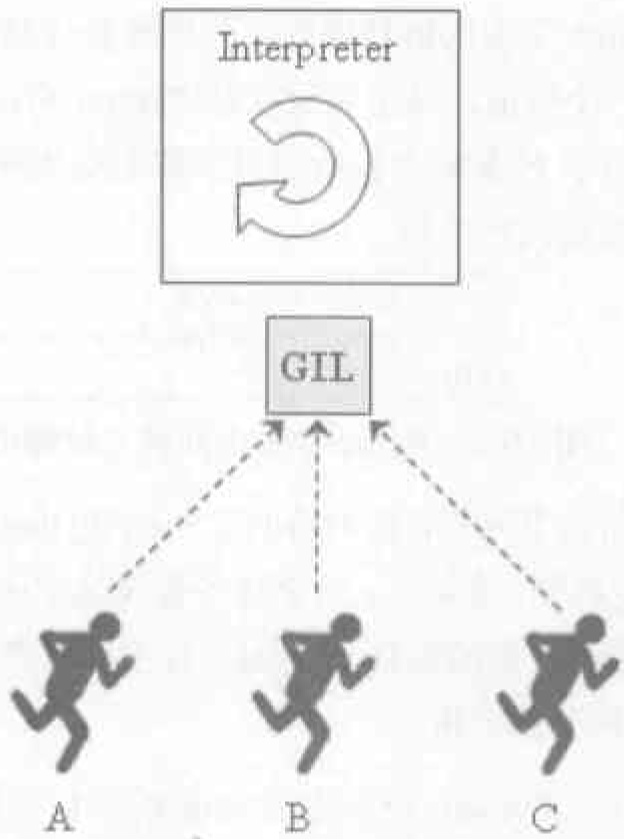



图 15-1 Python 线程机制的粗略模型

从之前的分析中，我们知道，对于 Python 而言，字节码解释器是 Python 的核心所在，所以 Python 通过 GIL 来互斥不同线程对解释器的使用。在图 15-1 中，三个拟人化的线程 A，B 和 C 都需要使用解释器来执行字节码，以完成某种计算，但是在这之前，它们必须获得 GIL，因为 GIL 把守着通往字节码解释器的大门。当某个线程（A）获得了 GIL 之后，其他的两个线程（B，C）只能等待 A 释放 GIL 之后，然后才能进入解释器，执行一些计算。实际上，Python 的 GIL 背后所保护的不仅仅是 Python 的解释器，同样还有 Python 的 C API，在 C/C++和 Python 的混合开发中，在涉及到原生线程和 Python 线程的相互协作时，也需要通过 GIL 进行互斥。关于这一点，我们将在后面详细阐述。

那么 A 在何时释放 GIL 呢？如果等到 A 使用完了解释器之后才释放 GIL，这也就意味着，并行的计算退化为了串行的计算，要这样的多线程机制有什么意义呢？所有毫无疑问的，Python 拥有一套线程的调度机制。

对于线程调度机制而言，同操作系统的进程调度一样，最关键的是要解决两个问题：

- 在何时挂起当前线程，选择处于等待状态的下一个线程？
- 在众多的处于等待状态的候选线程中，选择激活哪一个线程？

在 Python 的多线程机制中，这两个问题是分别由不同的层次解决的。对于何时进行线程调度的问题，是由 Python 自身决定的。考虑一下操作系统是如何进行进程的切换的，当一个进程执行了一段时间之后，发生了时钟中断，操作系统响应时钟中断，并在这时开始进行进程的调度。同样，Python 中也是通过软件模拟了这样的时钟中断，来激活线程的

调度。我们知道，Python 字节码解释器的工作原理是按照指令的顺序一条一条地顺序执行，Python 内部维护着一个数值，这个数值就是 Python 内部的时钟，如果这个数值为 N，则意味着 Python 在执行了 N 条指令以后应该立即启动线程调度机制，图 15-2 显示了如何获得 Python 内部默认设定的这个值。

```
>>> import sys
>>> sys.getcheckinterval()
100
```

图 15-2 Python2.5 内部的“时钟中断”间隔值

图 15-2 显示的结果意味着，在当前的 2.5 中，Python 的默认行为是在执行了 100 条指令以后启动线程调度机制。实际上，这个值不仅仅是用来进行线程调度的，在内部，Python 也使用它来检查是否有异步的事件（event）发生，需要处理。我们可以通过 `sys.setcheckinterval()`来调节这个值。

现在我们知道了，Python 控制着什么时候进行线程调度，当一个线程获得了访问 Python 解释器的所必须的 GIL 并进入 Python 解释器后，Python 内部的监测机制就开始启动，当这个线程执行了 100 条指令之后，Python 解释器将强制挂起当前线程，开始切换到下一个处于等待状态的进程。

那么究竟 Python 会在众多的等待线程中选择哪一个幸运儿呢？答案是，不知道。没错，对于这个问题，Python 完全没有插手，而是交给了底层的操作系统来解决。也就是说，Python 借用了底层操作系统所提供的线程调度机制来决定下一个进入 Python 解释器的线程究竟是谁。

这一点至关重要，这就意味着 Python 中的线程实际上就是操作系统所支持的原生线程，而并非如坊间所流传的那样：Python 的线程并非原生线程，而是模拟出来的。Python 中的多线程机制正是建立在操作系统的原生线程的基础之上，对应不同的操作系统，有不同的实现，然而最终，在各不相同的原生线程的基础之上，Python 提供了一套统一的抽象机制，给 Python 的使用者一个非常简单而方便的多线程工具箱，这就是 Python 中的两个 module：thread 以及在其之上的 threading。

## 15.2 初见 Python Thread

Python 所提供的最基础的多线程机制的接口是 `thread module`。这个 module 是一个 `builtin module`，用 C 实现。在 `thread module` 的基础上，Python 提供了一个更高层的多线程机制接口，即 `threding module`。`threading module` 是一个标准库中的 module，用 Python 语言实现，为用户提供了更方便的多线程机制接口。

我们的目标是要剖析 Python 中的多线程机制是如何实现的，而非学习在 Python 中如何进行多线程编程，所以重点会放在 thread module 上。通过这个 module，看一看 Python 对操作系统的原生线程机制所做的精巧的包装。

我们通过下面所示的 thread1.py 开始充满趣味的多线程之旅。

```
[thread1.py]
import thread
import time

def threadProc():
    print 'sub thread id : ', thread.get_ident()
    while True:
        print "Hello from sub thread ", thread.get_ident()
        time.sleep(1)

print 'main thread id : ', thread.get_ident()
thread.start_new_thread(threadProc, ())
while True:
    print "Hello from main thread ", thread.get_ident()
    time.sleep(1)
```

在 thread module 中，Python 向用户提供的多线程机制的接口其实可以说少得可怜，当然，也正因为如此，才使 Python 中的多线程编程变得非常的简单而方便。我们来看看在 thread module 的实现文件 threadmodule.c 中，thread module 为 Python 使用者提供的所有多线程机制接口。

```
[threadmodule.c]
static PyMethodDef thread_methods[] = {
    {"start_new_thread",
(PyCFunction)thread_PyThread_start_new_thread,…},
    {"start_new",    (PyCFunction)thread_PyThread_start_new_thread, …},
    {"allocate_lock",    (PyCFunction)thread_PyThread_allocate_lock, …},
    {"allocate",    (PyCFunction)thread_PyThread_allocate_lock, …},
    {"exit_thread",    (PyCFunction)thread_PyThread_exit_thread, …},
    {"exit",    (PyCFunction)thread_PyThread_exit_thread, …},
    {"interrupt_main", (PyCFunction)thread_PyThread_interrupt_main,…},
    {"get_ident",    (PyCFunction)thread_get_ident, …},
    {"stack_size",    (PyCFunction)thread_stack_size, …},
    {NULL,        NULL}        /* sentinel */
};
```

我们发现，thread module 中有的接口居然以不同的形式出现了两次，比如“start_new_thread”和“start_new”，实际上在 Python 内部，对应的都是 thread_PyThread_start_new_thread 这个函数。所以，thread module 所提供的接口，真的是少得可怜。在我们的 thread1.py 中我们使用了其中两个接口。关于这两个接口的详细介绍，请参阅 Python 文档。

如图 15-3 展示了 thread1.py 的运行结果。

```
F:\PythonBook\Src\thread>python thread1.py
main thread id : 3288
Hello from main thread 3288
sub thread id : 1132
Hello from sub thread 1132
Hello from main thread 3288
Hello from sub thread 1132
Hello from main thread 3288
Hello from sub thread 1132
......
```

图 15-3 thread1.py 的运行结果

# 15.3 Python 线程的创建

在 Python 的 thread module 所提供的接口中，一定不能少的肯定是创建线程的接口，倘若没有这个接口，生活还有什么意义呢☺？在上面的 thread1.py 中，我们正是通过其提供的 start_new_thread 创建了一个崭新的线程。好，我们就进入这个 start_new_thread，看看 Python 是如何进行创世纪的工作的（见代码清单 15-1）。

**代码清单 15-1**

```
[threadmodule.c]
static PyObject* thread_PyThread_start_new_thread(PyObject *self, PyObject
  *fargs)
{
    PyObject *func, *args, *keyw = NULL;
    struct bootstate *boot;
    long ident;

    PyArg_UnpackTuple(fargs, "start_new_thread", 2, 3, &func, &args,
      &keyw);
    //[1]: 创建 bootstate 结构
    boot = PyMem_NEW(struct bootstate, 1);
    boot->interp = PyThreadState_GET()->interp;
    boot->func = func;
    boot->args = args;
    boot->keyw = keyw;
    //[2]: 初始化多线程环境
    PyEval_InitThreads(); /* Start the interpreter's thread-awareness */
    //[3]: 创建线程
    ident = PyThread_start_new_thread(t_bootstrap, (void*) boot);
    return PyInt_FromLong(ident);
}
```

在 thread_PyThread_start_new_thread 中，Python 虚拟机通过三个主要的动作，完成一个线程的创建。

代码清单 15-1 的[1]、[2]和[3]分别有如下含义：

[1] 创建并初始化 bootstate 结构 boot，在 boot 中，将保存关于线程的一切信息，如：线程过程，线程过程的参数等。

[2] 初始化 Python 的多线程环境。

[3] 以 boot 为参数，创建操作系统的原生线程。

在代码清单 15-1 的[1]中，我们注意到 boot->interp 中保存了 Python 的 PyInterpreterState 对象，这个对象中携带了 Python 的 module pool 这样的全局信息，Python 中所有的 thread 都会共享这些全局信息。

关于代码清单 15-1 的[2]处所示的多线程环境的初始化动作，有一点需要特别说明，当 Python 启动时，是并不支持多线程的。换句话说，Python 中支持多线程的数据结构以及 GIL 都是没有创建的，Python 之所以有这种行为是因为大多数的 Python 程序都不需要多线程的支持。假如一个简单地统计词频的 Python 脚本中居然出现了多线程，面对这样的代码，我们一定都会抓狂的☺。

对多线程的支持并非是没有代价的，最简单的一点，如果激活多线程机制，而执行的 Python 程序中并没有多线程，那么在 100 条指令之后，Python 虚拟机同样会激活线程的调度。而如果不激活多线程，Python 虚拟机则不用做这些无用功。所以 Python 选择了让用户激活多线程机制的策略。在 Python 虚拟机启动时，多线程机制并没有被激活，它只支持单线程，一旦用户调用 thread.start_new_thread，明确指示 Python 虚拟机创建新的线程，Python 就能意识到用户需要多线程的支持，这个时候，Python 虚拟机会自动建立多线程机制需要的数据结构、环境以及那个至关重要的 GIL。

## 15.3.1 建立多线程环境

多线程环境的建立，说得直白一点，主要就是创建 GIL。我们已经知道 GIL 对于 Python 的多线程机制的重要意义，然而这个 GIL 到底是如何实现的呢，呃，这是一个很有趣的问题。

```
[pythread.h]
typedef void *PyThread_type_lock;

[ceval.c]
static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */
static long main_thread = 0;

void PyEval_InitThreads(void)
{
    if (interpreter_lock)
        return;
```

```
    interpreter_lock = PyThread_allocate_lock();
    PyThread_acquire_lock(interpreter_lock, 1);
    main_thread = PyThread_get_thread_ident();
}
```

终于见识到了神秘的 GIL（interpreter_lock），没想到吧，万万没想到，它居然指示一个简单的 void*。但是转念一想，在 C 中 void*几乎可以是任何东西，这家伙，可是个万能容器啊。

可以看到，无论创建多少个线程，Python 建立多线程环境的动作只会执行一次。在 PyEval_InitThreads 的开始，Python 会检查 GIL 是否已经被创建，如果是，则不再进行任何动作，否则，就会创建这个 GIL。创建 GIL 的工作由 PyThread_allocate_lock 完成，我们来看一看这个 GIL 到底是何方神圣。

```
[thread_nt.h]
PyThread_type_lock PyThread_allocate_lock(void)
{
    PNRMUTEX aLock;

    if (!initialized)
        PyThread_init_thread();
    aLock = AllocNonRecursiveMutex() ;
    return (PyThread_type_lock) aLock;
}
```

在这里，我们终于看到了 Python 中多线程机制的平台相关性，在 Python25\Python 目录下，有一大批 thread_***.h 这样的文件，在这些文件中，包装了不同操作系统的原生线程，并通过统一的接口暴露给 Python，比如这里的 PyThread_allocate_lock 就是这样一个接口。我们这里的 thread_nt.h 中包装的是 Win32 平台的原生 thread，在本章中后面的代码剖析中，还会有大量与平台相关的代码，我们都以 Win32 平台为例。

在 PyThread_allocate_lock 中，与 PyEval_InitThreads 非常类似的，它会检查一个 initialized 的变量，如果说 GIL 指示着 Python 的多线程环境是否已经建立，那么这个 initialized 变量就指示着为了使用底层平台所提供的原生 thread，必须的初始化动作是否完成。这些必须的初始化动作通常都是底层操作系统所提供的 API，不同的操作系统可能需要不同的初始化动作。在 Win32 平台下，不需要任何的初始化动作，所以 PyThread_init_thread 的唯一作用就是设置 initialized 变量：

```
[thread.c]
void PyThread_init_thread(void)
{
    if (initialized)
        return;
    initialized = 1;
    PyThread__init_thread();
}

[thread_nt.h]
static void PyThread__init_thread(void) { }
```

在 PyThread_allocate_lock 中，出现了一个关键的结构体 PNRMUTEX，我们发现，这个结构体是函数的返回值，实际上也就是 PyEval_InitThread 中需要创建的那个 interperter_lock（GIL）。原来 GIL 就是这个家伙，我们来看一看它的真身。

```
[thread_nt.h]
typedef struct NRMUTEX {
    LONG   owned ;
    DWORD  thread_id ;
    HANDLE hevent ;
} NRMUTEX, *PNRMUTEX ;
```

在 NRMUTEX 中，所有的数据成员的类型都是 Win32 平台下的类型风格了，owned 和 thread_id 都很普通，而其中的 HANDLE hevent 却值得注意，我们来看看 AllocNonRecursiveMutex 究竟为这个 hevent 准备了什么。

```
[thread_nt.h]
PNRMUTEX AllocNonRecursiveMutex(void)
{
    PNRMUTEX mutex = (PNRMUTEX)malloc(sizeof(NRMUTEX)) ;
    if(mutex && !InitializeNonRecursiveMutex(mutex)) {
            free(mutex);
            Mutex = NULL;
    }
    return mutex ;
}

BOOL InitializeNonRecursiveMutex(PNRMUTEX mutex)
{
    ……
    mutex->owned = -1 ;  /* No threads have entered NonRecursiveMutex */
    mutex->thread_id = 0 ;
    mutex->hevent = CreateEvent(NULL, FALSE, FALSE, NULL) ;
    return mutex->hevent != NULL ;  /* TRUE if the mutex is created */
}
```

一切真相大白了，原来，GIL（NRMUTEX）中的 hevent 就是 Win32 平台下的 Event 这个内核对象，而其中的 thread_id 将记录任一时刻获得 GIL 的线程的 id。

到了这里，Python 中的线程互斥机制的真相渐渐浮出水面，看来 Python 是通过 Win32 下的 Event 来实现了线程的互斥，熟悉 Win32 的朋友马上就可能想到，与这个 Event 对应的，必定有一个 WaitForSingleObject。

在 PyEval_InitThreads 通过 PyThread_allocate_lock 成功地创建了 GIL 之后，当前线程就开始遵循 Python 的多线程机制的规则：在调用任何 Python C API 之前，必须首先获得 GIL。因此 PyEval_InitThreads 紧接着通过 PyThread_acquire_lock 尝试获得 GIL。

```
[thread_nt.h]
int PyThread_acquire_lock(PyThread_type_lock aLock, int waitflag)
{
    int success ;
    success = aLock && EnterNonRecursiveMutex((PNRMUTEX) aLock, (waitflag
```

```
        == 1 ? INFINITE : 0)) == WAIT_OBJECT_0 ;
    return success;
}

DWORD EnterNonRecursiveMutex(PNRMUTEX mutex, BOOL wait)
{
    /* Assume that the thread waits successfully */
    DWORD ret;

    /* InterlockedIncrement(&mutex->owned) == 0 means that no thread
       currently owns the mutex */
    if (!wait)
    {
        if (InterlockedCompareExchange((PVOID *)&mutex->owned, (PVOID)0,
              (PVOID)-1) != (PVOID)-1)
            return WAIT_TIMEOUT ;
        ret = WAIT_OBJECT_0 ;
    }
    else
        ret = InterlockedIncrement(&mutex->owned) ?
            /* Some thread owns the mutex, let's wait... */
            WaitForSingleObject(mutex->hevent, INFINITE) : WAIT_OBJECT_0 ;

    mutex->thread_id = GetCurrentThreadId() ; /* We own it */
    return ret ;
}
```

PyThread_acquire_lock 有两种工作方式，通过函数参数 waitflag 来区分。这个 waitflag 指示当 GIL 当前不可获得时，是否进行等待，更直接地说，就是当前线程是否通过 WaitForSingleObject 将自身挂起，直到别的线程释放 GIL，然后由操作系统将自己唤醒。

如果 waitflag 为 0，Python 会检查当前 GIL 是否可用，GIL 中的 owned 是指示 GIL 是否可用的变量，在前面的 InitializeNonRecursiveMutex 中我们看到这个值被初始化为-1，Python 会检查这个值是否为-1，如果是，则意味着 GIL 可用，必须将其置为 0，当 owned 为 0 后，表示该 GIL 已经被一个线程占用，不再可用。对于我们这里分析的调用 PyEval_InitThread 的主线程而言，由于在初始化 GIL 之后就调用 PyThread_acquire_lock 申请 GIL，到这时，并没有第二个线程被创建，所以主线程会轻而易举地获得 GIL 的使用权。

注意这里的检查和更新 owned 的操作是通过一个 Win32 的系统 API——InterlockedCompareExchange——来完成的。这个 API 是一个原子操作，其函数原形和功能如下。

```
原形: InterlockedCompareExchange(PLONG dest, long exchange, long compared)
功能: 如果*dest == compared,那么*dest = exchange
```

与 InterlockedCompareExchange 相同的，InterlockedIncrement 也是一个原子操作，其功能是将 mutex->owned 的值增加 1。从这里可以看到，当一个线程开始等待 GIL 时，其 owned 就会被增加 1。显然我们可以猜测，当一个线程最终释放 GIL 时，一定会将

GIL 的 owned 减 1，这样当所有需要 GIL 的线程都最终释放了 GIL 之后，owned 会再次变为-1，意味着 GIL 再次变为可用。

为了清晰地展示这一点，我们现在就来看看 PyThread_aquire_lock 的逆运算，PyThread_release_lock 每一个将从运行转态转为等待状态的线程都会在被挂起之前调用它以释放对 GIL 的占有。

```
[thread_nt.h]
void PyThread_release_lock(PyThread_type_lock aLock)
{
   LeaveNonRecursiveMutex((PNRMUTEX) aLock);
}

BOOL LeaveNonRecursiveMutex(PNRMUTEX mutex)
{
   /* We don't own the mutex */
   mutex->thread_id = 0 ;
   return
      InterlockedDecrement(&mutex->owned) < 0 ||
      SetEvent(mutex->hevent) ; /* Other threads are waiting, wake one on
        them up */
}
```

最终，一个线程在释放 GIL 时，会通过 SetEvent 通知所有在等待 GIL 的 hevent 这个 Event 内核对象的线程，结合前面的分析，如果这时候有线程在等待 GIL 的 hevent，那么将被操作系统唤醒。这就是我们在前面介绍的 Python 将线程调度的第二个难题委托给操作系统来实现的机制。

到了这时，调用 PyEval_InitThread 的线程（也就是 Python 主线程）已经成功获得了 GIL，最后会调用 PyThread_get_thread_ident()，通过 Win32 的 API：GetCurrent-ThreadId，获得当前 Python 主线程的 id，并将其赋给 main_thread，main_thread 是一个静态全局变量，专职存储 Python 主线程的线程 id，用以判断一个线程是否是 Python 主线程。

最后，我们在图 15-4 中给出整个 PyEval_InitThread 的函数调用关系。

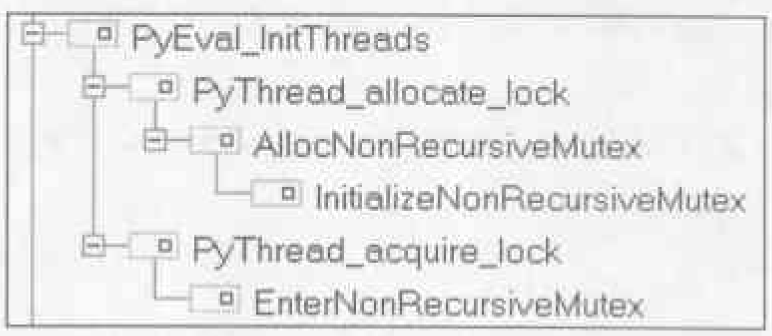



图 15-4 PyEval_InitThreads 中的函数调用关系

## 15.3.2 创建线程

### 15.3.2.1 子线程的诞生

在完成了多线程环境的初始化之后，Python 会开始创建底层平台的原生 thread，以 thread1.py 为例，这个原生 thread 将执行 `threadProc` 所定义的操作。从现在开始，为了描述的清晰性，我们将 Python 主线程，也就是调用 `thread_PyThread_start_new_thread` 创建新的线程的线程称为主线程，而将与 `threadProc` 对应的原生 thread 称之为子线程。现在我们来看看一个子线程是如何被创建的（见代码清单 15-2）。

**代码清单 15-2**

```
[threadmodule.c]
static PyObject* thread_PyThread_start_new_thread(PyObject *self, PyObject
  *fargs)
{
    PyObject *func, *args, *keyw = NULL;
    struct bootstate *boot;
    long ident;

    PyArg_UnpackTuple(fargs, "start_new_thread", 2, 3, &func, &args, &keyw);
    //[1]: 创建 bootstate 结构
    boot = PyMem_NEW(struct bootstate, 1);
    boot->interp = PyThreadState_GET()->interp;
    boot->func = func;
    boot->args = args;
    boot->keyw = keyw;
    //[2]: 初始化多线程环境
    PyEval_InitThreads(); /* Start the interpreter's thread-awareness */
    //[3]: 创建线程
    ident = PyThread_start_new_thread(t_bootstrap, (void*) boot);
    return PyInt_FromLong(ident);

[thread.c]
/* Support for runtime thread stack size tuning.
   A value of 0 means using the platform's default stack size
   or the size specified by the THREAD_STACK_SIZE macro. */
static size_t _pythread_stacksize = 0;

[thread_nt.h]
long PyThread_start_new_thread(void (*func)(void *), void *arg)
{
    unsigned long rv;
    callobj obj;

    obj.id = -1;    /* guilty until proved innocent */
    obj.func = func;
    obj.arg = arg;
    obj.done = CreateSemaphore(NULL, 0, 1, NULL);

    rv = _beginthread(bootstrap, _pythread_stacksize, &obj); /* use default stack size */
```

```
    if (rv == (unsigned long)-1) {
        //创建 raw thread 失败
        obj.id = -1;
    }
    else {
        WaitForSingleObject(obj.done, INFINITE);
    }
    CloseHandle((HANDLE)obj.done);
    return obj.id;
}
```

主线程通过调用 PyThread_start_new_thread 完成创建子线程的工作。为了清晰地理解 PyThread_start_new_thread 的工作，我们需要特别注意该函数的参数，从 thread_PyThread_start_new_thread 中可以看到，这里的 func 实际上是函数 t_bootstrap，而 arg 则是在 thread_PyThread_start_new_thread 中创建的 bootstate 结构体 boot。在 boot 中，保存着 Python 程序（thread1.py）中所定义的线程的信息。

PyThread_start_new_thread 首先会将 func 和 arg 都打包到一个类型为 callobj 的结构体 obj 中，我们来看看这个 obj。

```
[thread_nt.h]
typedef struct {
    void (*func)(void*);
    void *arg;
    long id;
    HANDLE done;
} callobj;
```

图 15-5 显示了对应 thread1.py 所构造出的 callobj 对象的示意图。

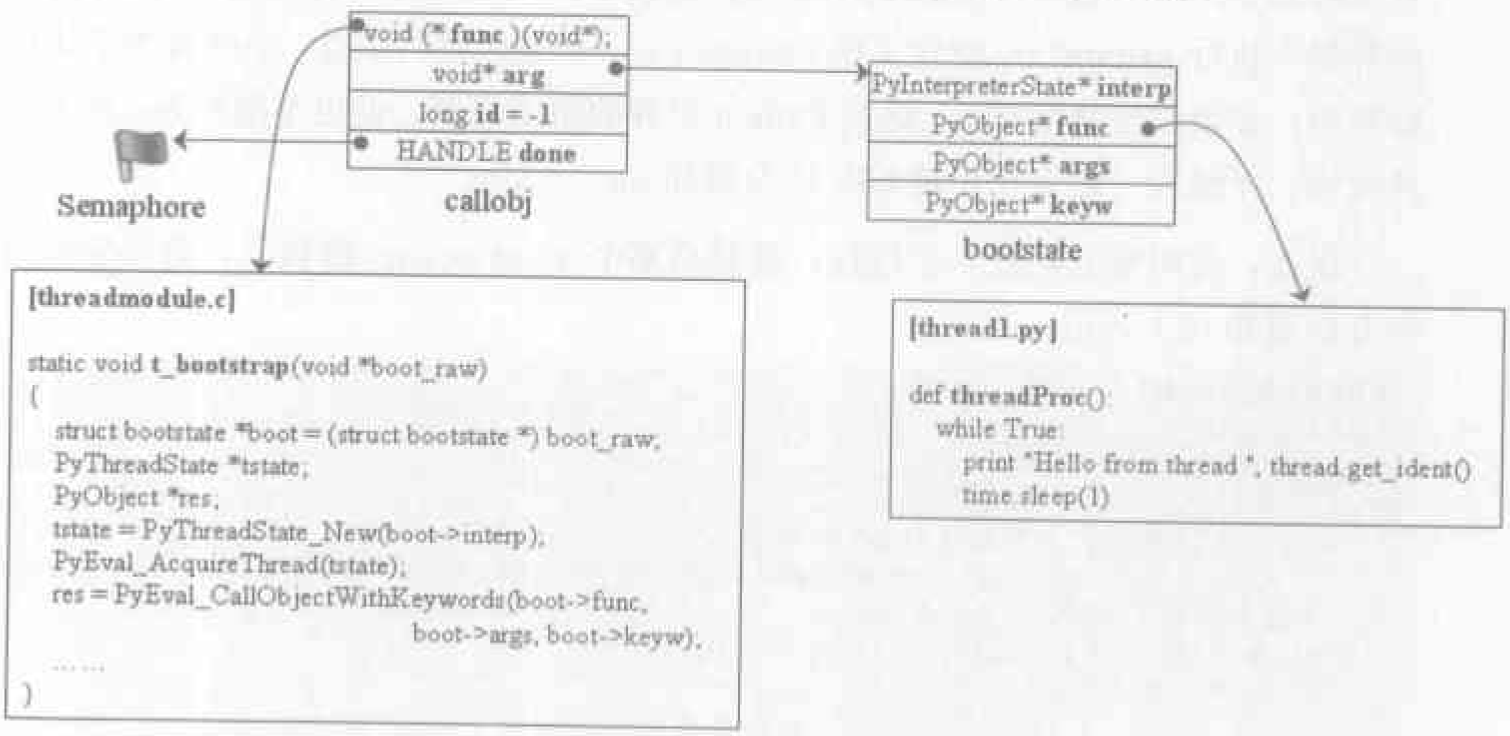



图 15-5 thread1.py 对应的 callobj 对象的示意图

值得注意的是，obj.done 是一个 Win32 下的 Semaphore 内核对象，这个特殊的内核对象的用途我们马上就会看到。我们创建线程的工作需要 func 和 arg，但是 Win32 下创

建线程的 API 只允许用户指定一个自定义的参数，这就是需要用 obj 来打包的原因。

完成打包之后，调用 Win32 下创建 thread 的 API：_beginthread 来完成线程的创建。奇怪的是，我们期望的线程过程应该是 thread1.py 中定义的那个 threadPoc 呀，而这里指定的线程过程却是一个相当面生的 bootstrap。实际上，在 bootstrap 中，会最终调用 thread1.py 中定义的 threadProc。

但是，这里有一个至关重要的转折，还记得我们现在在哪里吗？没错，我们现在是沿着主线程的执行路径在剖析，而对 bootstrap 的调用并不是在主线程中发生的，而是在通过_beginthread 所创建的子线程中发生的。从这里开始，我们需要特别注意代码的执行是在哪个线程中执行的，这对于理解 Python 的多线程机制相当重要。

好了，花开两朵，各表一枝。我们继续沿着主线程的执行路径前进。如果不出什么意外，_beginthread 将最终成功地创建 Win32 下的原生线程，并顺利返回。在返回之后，主线程开始将自己挂起，等待 obj.done。我们前面看到，这是一个 Win32 的 Semaphore 内核对象。由于 obj 已经作为参数传递给了子线程，所以我们猜想，子线程会设置这个 Semaphore，并最终唤醒主线程。

现在我们来理清一下 Python 当前的状态。Python 当前实际上由两个 Win32 下的原生 thread 构成，一个是执行 python 程序（python.exe）时操作系统创建的主线程，另一个是我们通过 thread1.py 创建的子线程。主线程在执行 PyEval_InitThread 的过程中，获得了 GIL，但是目前已经被挂起，这是为了等待子线程中控制着的 obj.done。子线程的线程过程是 bootstrap，不过我们刚才已经猜测了，从 bootstrap 出发，最终将在 Python 解释器中执行 python1.py 中定义的 theadProc。但是，我们知道，子线程为了访问 Python 解释器，必须首先获得 GIL，这是 Python 世界的游戏规则，谁也不能例外。所以，为了避免死锁，子线程一定会在申请 GIL 之前通知 obj.done。

现在，是时候开始进入子线程，也就是那个 bootstrap 函数了，看看它究竟什么时候开始通知 obj.done。

```
[thread_nt.h]
static int
bootstrap(void *call)
{
    callobj *obj = (callobj*)call;
    /* copy callobj since other thread might free it before we're done */
    //这里将得到函数 t_bootstrap
    void (*func)(void*) = obj->func;
    void *arg = obj->arg;

    obj->id = PyThread_get_thread_ident();
    ReleaseSemaphore(obj->done, 1, NULL);
    func(arg);
    return 0;
}
```

在 bootstrap 中，子线程完成了三个动作：

- 获得线程 id；
- 通知 obj->done 内核对象；
- 调用 t_bootstrap；

在这里我们看到，子线程在申请 GIL 之前确实通知了前面提到的 obj.done 内核对象以唤醒主线程。那么，Python 为什么需要让主线程等待子线程的通知呢，在这里一切都明了了。原来，主线程所调用的 PyThread_start_new_thread 需要返回所创建的子线程的线程 id，然而子线程的线程 id 只有在子线程被激活后才能在子线程中获取，因此主线程等的就是这个子线程 id，一旦子线程设置了 obj->id，就会设法唤醒主线程。

从这里开始，主线程和子线程开始分道扬镳，主线程在返回子线程 id 之后，会继续执行后续的字节码，因为我们知道，这时候，主线程手里握着 GIL。而子线程则将进入 t_bootstrap，并最终进入等待 GIL 的状态。

```
[threadmodule.c]
static void t_bootstrap(void *boot_raw)
{
    struct bootstate *boot = (struct bootstate *) boot_raw;
    PyThreadState *tstate;
    PyObject *res;

    tstate = PyThreadState_New(boot->interp);
    PyEval_AcquireThread(tstate);
    res = PyEval_CallObjectWithKeywords(boot->func, boot->args,
boot->keyw);
    PyMem_DEL(boot_raw);
    PyThreadState_Clear(tstate);
    PyThreadState_DeleteCurrent();
    PyThread_exit_thread();
}
```

子线程从这里开始了与主线程对 GIL 的竞争。在 t_bootstrap 中，所进行的第一个动作——PyEval_AcquireThread——就是申请 GIL，当 PyEval_AcquireThread 结束之后，子线程也就获得了 GIL，并且做好了一切执行的准备。接下来子线程通过 PyEval_CallObjectWithKeywords，将最终调用我们已经非常熟悉的 PyEval_EvalFrameEx，也就是 Python 的字节码执行引擎。传递进 PyEval_CallObjectWithKeywords 的 boot->func 是一个 PyFunctionObject 对象，正是 therad1.py 中定义的 threadProc 编译后的结果。

在 PyEval_CallObjectWithKeywords 结束之后，子线程将释放 GIL，并完成销毁线程的所有扫尾工作，到了这里，子线程就结束了。

从 t_bootstrap 的代码看上去，似乎子线程会一直执行，直到子线程的所有计算都完成，才会通过 PyThreadState_DeleteCurrent 释放 GIL。如此一来，那主线程岂非一

直都会处于等待 GIL 的状态？如果真是这样，那 Python 显然就不可能支持多线程机制了。实际上在 PyEval_EvalFrameEx 中，. 图 15-2 中显示的 Python 内部维护的那个模拟时钟中断会不断地激活线程的调度机制，在子线程和主线程之间不断地进行切换，从而真正实现多线程机制。当然，这一点我们将在后面详细剖析。现在我们感兴趣的是子线程在 PyEval_AcquireThreade 中到底做了什么。

```
[ceval.c]
void PyEval_AcquireThread(PyThreadState *tstate)
{
    if (tstate == NULL)
        Py_FatalError("PyEval_AcquireThread: NULL new thread state");
    //检查 interpreter_lock，确保已经调用 PyEval_InitThreads 并创建了 GIL
    assert(interpreter_lock);
    //获得 GIL
    PyThread_acquire_lock(interpreter_lock, 1);
    //在 PyThreadState_Swap 中更新指向"当前线程"的
    //线程状态对象指针_PyThreadState_Current
    if (PyThreadState_Swap(tstate) != NULL)
        Py_FatalError("PyEval_AcquireThread: non-NULL old thread state");
}

[pystate.c]
PyThreadState * PyThreadState_Swap(PyThreadState *newts)
{
    PyThreadState *oldts = _PyThreadState_Current;
    _PyThreadState_Current = newts;
    return oldts;
}
```

到这里，了解了 PyEval_AcquireThread，似乎创建线程的机制都清晰了。但实际上，有一个非常重要的机制——线程状态保护机制——隐藏在了一个毫不起眼的地方：PyThreadState_New。这个机制对于理解 Python 中线程的创建和维护是非常关键的。

### 15.3.2.2 线程状态保护机制

要剖析线程状态的保护机制，我们首先需要回顾一下线程状态。在 Python 中，每一个 Python 线程都会有一个线程状态对象与之关联，在线程状态对象中，记录了每一个线程所独有的一些信息。实际上，在剖析 Python 的初始化过程时，我们曾经见过这个对象。

```
[pystate.h]
typedef struct _ts {
    struct _ts *next;
    PyInterpreterState *interp;
    struct _frame *frame;
    int recursion_depth;
    ......
    int gilstate_counter;
    long thread_id; /* Thread id where this tstate was created */
} PyThreadState;
```

每一个线程对应的线程状态对象都保存着这个线程当前的 PyFrameObject 对象，线程的 id 这样一些信息。有时候，线程是需要访问这些信息的。比如考虑一个最简单的情形，在某种情况下，每个线程都需要访问线程状态对象中所保存的 thread_id 信息，显然，线程 A 获得的应该是 A 的 thread_id，线程 B 亦然。倘若线程 A 获得的是 B 的 thread_id，那就坏菜了。这就意味着 Python 内部必须有一套机制，这套机制与操作系统管理进程的机制非常类似。我们知道，在操作系统从进程 A 切换到进程 B 时，首先会保存进程 A 的上下文环境，再进行切换；当从进程 B 切换回进程 A 时，又会恢复进程 A 的上下文环境，这样就保证了进程 A 始终是在属于自己的上下文环境中运行。

这里的线程状态对象就等同于进程的上下文，Python 同样会有一套存储、恢复线程状态对象的机制。同时，在 Python 内部，维护着一个全局变量：PyThreadState * _PyThreadState_Current。当前活动线程所对应的线程状态对象就保存在这个变量里，当 Python 调度线程时，会将被激活的线程所对应的线程状态对象赋给_PyThreadState_Current，使其始终保存着活动线程的状态对象。

这就引出了这样的一个问题：Python 如何在调度进程时，获得被激活线程对应的状态对象？Python 内部会通过一个单向链表来管理所有的 Python 线程的状态对象，当需要寻找一个线程对应的状态对象时，就遍历这个链表，搜索其对应的状态对象。在此后的描述中，我们将这个链表称为“状态对象链表”。

下面我们来看一看实现这个机制的关键数据结构。

```
[thread.c]
struct key {
    struct key *next;
    long id;
    int key;
    void *value;
};

static struct key *keyhead = NULL;
```

图 15-6 展示了在运行时这个状态对象链表的示意图。

在 Python 中，对于这个状态对象链表的访问，不必在 GIL 的保护下进行。因为对于这个状态对象链表，Python 会创建一个独立的锁，专职对状态对象链表进行保护。这个锁的创建是在 Python 进行初始化的时候完成的。

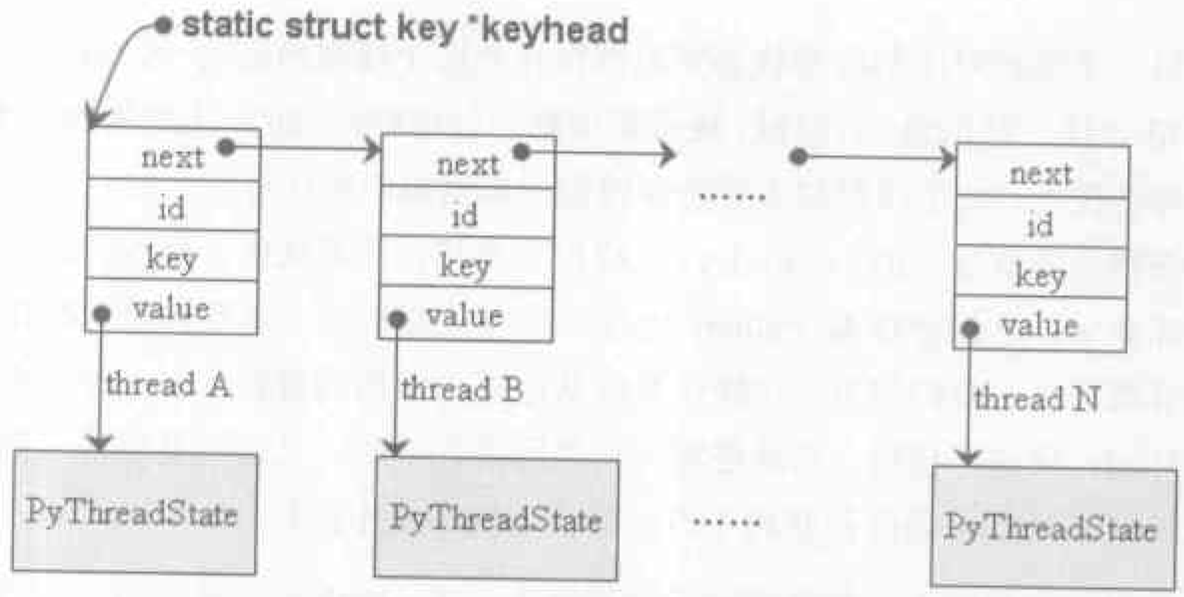



图 15-6 线程状态对象链表

```
[pystate.c]
static PyInterpreterState *autoInterpreterState = NULL;
static int autoTLSkey = 0;

void _PyGILState_Init(PyInterpreterState *i, PyThreadState *t)
{
    autoTLSkey = PyThread_create_key();
    autoInterpreterState = i;
    /* Now stash the thread state for this thread in TLS */
    assert(PyThread_get_key_value(autoTLSkey) == NULL);
    _PyGILState_NoteThreadState(t);
    ......
}

static void _PyGILState_NoteThreadState(PyThreadState* tstate)
{
    if (!autoTLSkey)
        return;
    PyThread_set_key_value(autoTLSkey, (void *)tstate);
    tstate->gilstate_counter = 1;
}

[thread.c]
static PyThread_type_lock keymutex = NULL;

int PyThread_create_key(void)
{
    if (keymutex == NULL)
        keymutex = PyThread_allocate_lock();
    return ++nkeys;
}
```

PyThread_create_key 将创建一个新的 key。注意，这里的 key 都是一个整数，而且，当 PyThread_create_key 第一次被调用时（在_PyGILState_Init 中的调用正是第一次调用），会通过 PyThread_allcate_lock 创建一个 keymutex。根据我们前面的分析，这个 keymutex 实际上和 GIL 一样，都是一个 PNRMUTEX 结构体，而在这个结构体中，维护着一个 Win32 下的 Event 内核对象。这个 keymutex 的功能就是用来互斥对状态对象链表的访问。

在_PyGILState_Init 中，创建的新 key 被 Python 维护的全局变量 autoTLSkey 接收，其中的 TLS 是 Thread Local Store 的缩写，这个 autoTLSkey 将用作 Python 保存所有线程的状态对象的一个参数，即是图 15-6 中的 key 值。也就是说，状态对象列表中所有 key 结构体中的 key 值都会是 autoTLSkey。哎，那位看官说了，你看 PyThread_create_key 返回的是 nkeys 的递增后的值啊，就是说每 create 一次，得到的结果都是不同的，怎么能说所有的 key 都是一样的呢？事实上，在整个 Python 的源码中，PyThread_create_key 只在_PyGILState_Init 中被调用了，而这个_PyGILState_Init 只会在 Python 运行时环境初始化时调用一次。

那么如何区分哪个线程对应哪个状态对象呢，别忘了，我们还有线程 id 呢。图 15-6 中的 id 存储的正是各个线程的 id，根据这个 id，显然可以区分不同的线程了。

那么图 15-6 中的 key 看上去就有点多此一举了，实际上，图 15-6 中所示的链表结构并非是纯的状态对象链表，当在一个 key 结构体的 value 域存储的不是线程的状态对象，而是与线程相关的其他对象时，这个 key 值就有意义了。假如我们将一种状态对象设为 s，而另一种对象设为 o，在图 15-6 所示的链表中，存在着两个与某个线程 A 相关的 key 结构体。显然，对于这两个 key 结构体，id 域是完全一致的，那么当我们需要从这个链表中取出对象 o，而并非 s 时，该用什么来区分 o 和 s 呢？正是这个 key 值。所以实际上在 Python 中，与每个线程相关的对象可能有多种，而每一种对象都会对应一个 key 值，这个 key 值将会被所有的线程在存储这种对象时共享。对于我们这里关注的线程状态对象，其 key 值就是 autoTLSkey。同样，由于我们这里仅仅关注 Python 的线程机制，所以我们在后面的描述中还是将图 15-6 中的链表称为线程状态对象链表。

Python 提供了一些列操作状态对象链表的接口，其中核心是 find_key，见代码清单 15-3。

**代码清单 15-3**

```
[thread.c]
static struct key* find_key(int key, void *value)
{
    struct key *p;
    //[1]: 获得当前线程的线程 id, 并锁住线程状态对象链表
    long id = PyThread_get_thread_ident();
    PyThread_acquire_lock(keymutex, 1);
    //[2]: 遍历线程状态对象链表, 寻找 key 和 id 都匹配的元素
    for (p = keyhead; p != NULL; p = p->next) {
        if (p->id == id && p->key == key)
            goto Done;
    }
    //[3]: 如果[2]处的搜索失败, 则创建新的元素, 并加入线程状态对象链表
    p = (struct key *)malloc(sizeof(struct key));
    if (p != NULL) {
        p->id = id;
```

```
        p->key = key;
        p->value = value;
        p->next = keyhead;
        keyhead = p;
    }
Done:
    //[4]: 释放锁住的线程状态对象链表
    PyThread_release_lock(keymutex);
    return p;
}
```

虽然这个核心函数的名字叫 find_key，然而我们可以看到，它的作用并不仅仅是搜索，而且还包含了创建的动作。在代码清单 15-3 的[2]处，find_key 会遍历状态对象列表，搜索 key 和 id 都匹配的 key 结构体，如果搜索成功，则直接返回；而当搜索失败时，find_key 会在代码清单 15-3 的[3]处创建一个新的 key 结构体，并设置其中的 id，key 和 value，最后将其插入到状态对象列表的头部。

在代码清单 15-3 的[1]和[4]处我们看到了 Python 确实通过在_PyGILState_Init 中创建的 keymutex 来互斥对状态对象列表的访问。

在了解了这个核心函数之后，Python 为状态对象列表所提供的接口就显得非常清晰了。其实，就是简单的链表的插入、删除和查询操作。

```
[thread.c]
//查询操作
void* PyThread_get_key_value(int key)
{
    struct key *p = find_key(key, NULL);
    return p->value;
}

//插入操作
int PyThread_set_key_value(int key, void *value)
{
    struct key *p = find_key(key, value);
    return 0;
}

//删除操作
void PyThread_delete_key(int key)
{
    struct key *p, **q;

    PyThread_acquire_lock(keymutex, 1);
    q = &keyhead;
    while ((p = *q) != NULL) {
        if (p->key == key) {
            *q = p->next;
            free((void *)p);
        }
        else
            q = &p->next;
```

```
    }
    PyThread_release_lock(keymutex);
}
```

#### 15.3.2.3 从 GIL 到字节码解释器

现在，回过头来看一看刚才提到的 PyThreadState_New：

```
[pystate.c]
PyThreadState *
PyThreadState_New(PyInterpreterState *interp)
{
    PyThreadState *tstate = (PyThreadState *)malloc(sizeof(PyThreadState));
    ......
#ifdef WITH_THREAD
    _PyGILState_NoteThreadState(tstate);
#endif
    ......
    return tstate;
}
```

子线程在创建了自身的线程状态对象后，会通过_PyGILState_NoteThreadState 将这个对象放入到线程状态对象链表中去。

这里有一个需要特别注意的地方，即当前活动的 Python 线程不一定是获得了 GIL 的线程。正如我们在这里所展示的，在 thread1.py 中，主线程现在是获得了 GIL 的，但是子线程到现在还没有申请 GIL，自然也不会将自身挂起。由于主线程和子线程都是 Win32 的原生线程，所以操作系统可能在主线程和子线程之间切换。我们在这里要着重指出操作系统级的线程调度和 Python 级的线程调度是不同的。Python 级的线程调度一定意味着 GIL 拥有权的易手，而操作系统级的线程调度并不一定意味着 GIL 的易手，当所有的线程都完成了初始化动作之后，操作系统的线程调度和 Python 的线程调度才会同一。那时，Python 的线程调度会迫使当前活动线程释放 GIL，而这一操作会触发 GIL 中维护的 Event 内核对象，这个触发又进而触发操作系统的线程调度。而在线程的初始化完成之前，在 Python 线程调度和操作系统线程调度之间并没有这样的因果关系。图 15-7 中显示了 GIL 在 Python 级线程调度与操作系统级线程调度之间所起的桥梁作用。




图 15-7 Python 级的线程调度与操作系统级的线程调度

很显然，对于这个例子，子线程还没有获得 GIL。所以在 PyThreadState_New 之后，子线程开始争夺话语权了。

```
[threadmodule.c]
static void t_bootstrap(void *boot_raw)
```

```
{
    struct bootstate *boot = (struct bootstate *) boot_raw;
    PyThreadState *tstate;
    PyObject *res;

    tstate = PyThreadState_New(boot->interp);
    PyEval_AcquireThread(tstate);
    res = PyEval_CallObjectWithKeywords(boot->func, boot->args,
boot->keyw);
    ……
}
```

前面我们已经剖析过 PyEval_AcquireThread 的代码，在 PyEval_AcquireThread 中，子线程进行了最后的冲刺，它要生存，要执行，于是它开始通过 PyThread_acquire_lock 争取 GIL。到了这一步，子线程将自己挂起，操作系统的线程调度机制再也不能靠自身的力量将其唤醒，只有等待 Python 的线程调度机制强迫主线程放弃 GIL 后，子线程才会被唤醒；而子线程被唤醒之后，主线程却又陷入了苦苦地等待中，同样苦苦地守望着 Python 强迫子线程放弃 GIL 的那一刻。

当子线程被 Python 的线程调度机制唤醒之后，它所作的第一件事就是通过 PyThreadState_Swap 将 Python 维护的当前线程状态对象设置为其自身的状态对象，一如操作系统的进程上下文环境恢复一样。

现在我们的子线程开始等待 GIL，但是注意，线程的初始化还没有真正完成，因为子线程还没有顺利进入字节码解释器。当 Python 线程调度将子线程唤醒之后，子线程将回到 t_bootstrap 中，并进入 PyEval_CallObjectWithKeywords，从这里一直往前，最终将调用 PyEval_EvalFrameEx，进入解释器。到了那个时候，子线程和主线程一样，就完全被 Python 线程调度机制所控制了。

图 15-8 展示了从主线程开始创建子线程，到子线程进入 Python 解释器的所有函数调用。

需要注意的是，PyThread_start_new_thread 是在主线程中执行的，而从 bootstrap 开始，则是在子线程中执行的。其中涉及线程销毁的动作，如 PyThreadState_DeleteCurrent 等，将在后续的部分剖析。

到了这里，读者可能有些疑惑了，我们花费了大量篇幅剖析的线程状态对象链表似乎没有什么用啊。其实不然，试想一下，当线程调度发生时，在 Python 一级，需要通过之前剖析过的 PyTrheadState_Swap 函数切换当前的线程状态对象，这时候就需要根据线程 id 从线程状态对象链表中获取线程对象了。事实上，在 Python 内部的许多 API 中，比如 PyGILState_Ensure 等等中，都会涉及这个链表，这些 API 在 C 与 Python 交互时可能被大量调用，有兴趣的读者可以自行深入探索一下。

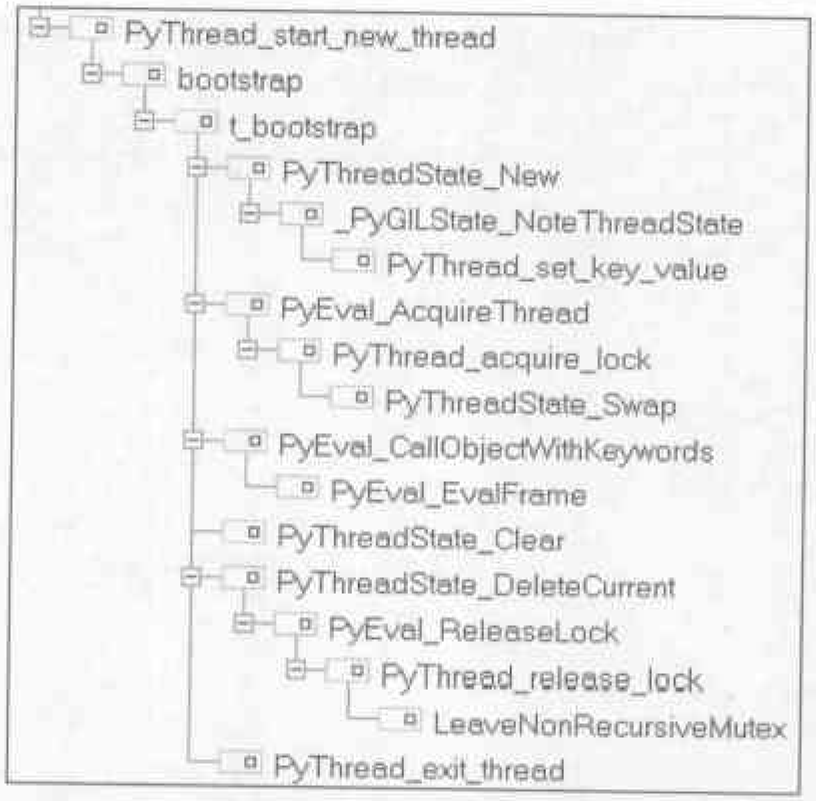



图 15-8 PyThread_start_new_thread 中的执行序列

## 15.4 Python 线程的调度

### 15.4.1 标准调度

当主线程和子线程都进入了 Python 解释器之后，Python 的线程之间的切换就完全由 Python 的线程调度机制掌控。Python 的线程调度机制是内建在 Python 的解释器核心 PyEval_EvalFrameEx 中的。在分析 Python 字节码解释器的框架时，我们曾给出过一个 PyEval_EvalFrameEx 的框架结构，但是在那里，并没有给出线程调度机制的实现，下面列出的是加入了线程调度机制的 PyEval_EvalFrameEx 的框架结构（见代码清单 15-4）。

**代码清单 15-4**

```
[ceval.c]
/* Interpreter main loop */
PyObject* PyEval_EvalFrameEx(PyFrameObject *f)
{
    ……
    why = WHY_NOT;


    for (;;) {
    ……
        if (--_Py_Ticker < 0) {
            //在切换线程之前，重置_Py_Ticker 为 100，为下一个线程做准备
            _Py_Ticker = _Py_CheckInterval;
            tstate->tick_counter++;
            if (interpreter_lock) {
```

```
            //[1]: 撤销当前线程状态对象，释放GIL，给别的线程一个机会
            PyThreadState_Swap(NULL);
            PyThread_release_lock(interpreter_lock);
            //[2]: 别的线程现在已经开始执行了，咱们重新再申请GIL，等待下一次被调度
            PyThread_acquire_lock(interpreter_lock, 1);
            PyThreadState_Swap(tstate) != NULL;
        }
    }
  fast_next_opcode:
      f->f_lasti = INSTR_OFFSET();
      /* Extract opcode and argument */
      opcode = NEXTOP();
      oparg = 0;   /* allows oparg to be stored in a register because
          it doesn't have to be remembered across a full loop */
      if (HAS_ARG(opcode))
          oparg = NEXTARG();
  dispatch_opcode:
      switch (opcode) {

      case NOP:
          goto fast_next_opcode;

      case LOAD_FAST:
          ……
      }
}
```

其中_Py_Ticker 就是 Python 内部通过软件模拟实现的时钟中断机制，而_Py_CheckInterval 保存着 Python 设定的指令数，这也就是通过 sys.getcheckinterval() 得到的值。在初始化时_Py_Ticker 和_Py_CheckInterval 是一样的，都是 100。

```
[ceval.c]
int _Py_CheckInterval = 100;
volatile int _Py_Ticker = 100;
```

PyEval_EvalFrameEx 每执行一条字节码指令，_Py_Ticker 就将减少 1；当执行了_Py_CheckInterval 条指令之后，_Py_Ticker 将减少到 0，这就将进入线程调度。注意，主线程和子线程都将调用 PyEval_EvalFrameEx，所以这里的描述可能会引起混乱，为了保证读者清晰地了解线程调度时所发生的一切，我们先来回忆一下 thread1.py 现在所处的情景。

主线程获得了 GIL，并且正在执行 PyEval_EvalFrameEx 函数的代码，这时子线程在 t_bootstrap 中调用 PyEval_AcquireThread。通过调用 PyThrad_acquire_lock 申请 GIL，但是由于 GIL 被主线程占有，所以子线程被挂起。

现在整个 Python 的进程中，只有一个活动的线程：主线程。主线程不断执行字节码，_Py_Ticker 的值不断减少，当_Py_Ticker 的值减少到 0 时，主线程在代码清单 15-4 的[1]处首先将维护当前线程状态对象的_PyThreadState_Current 设置为 NULL，然后释放掉 GIL。注意啦，注意啦，转折点到了：这时，由于等待 GIL 而被挂起的子线程被操作系

统的线程调度机制唤醒，从而最终进入 PyEval_EvalFrameEx。对于主线程，注意，虽然这时它已经失去了 GIL，但是由于它没有被挂起，所以对于操作系统的线程调度机制，它是可以被再次切换为活动线程的。

当操作系统的调度机制将主线程切换为活动线程之后，主线程将执行代码清单 15-4 的[2]，在 PyThread_acquire_lock 中，主线程申请 GIL，由于被子线程占有，主线程将自身挂起。从这时开始，操作系统的线程调度不能再将主线程切换为活动进行，除非等到子线程释放 GIL 之后。而子线程进入 PyEval_EvalFrameEx 之后，开始如之前主线程一般的行为，在执行了_Py_CheckInterval 条指令之后，子线程将执行代码[1]，释放 GIL，由此唤醒主线程。而子线程在继续执行[2]时，又将因等待 GIL 将自身挂起，如此反复，直至永恒，从而完整地在 Python 中实现了对多线程机制的支持。

有一点需要特别注意，实际上，Python 并非在执行了_Py_CheckInterval 条指令之后开始线程调度。从 PyEval_EvalFrameEx 的代码框架以及本书第 2 部分对 Python 虚拟机的剖析可以看到，很多字节码执行之后都会通过 goto 转移到 fast_next_opcode 处继续执行，这时是不会更新_Py_Ticker 的值的；而有的字节码在执行之后则会转移到最外层的 for 循环，这时是会更新_Py_Ticker 的。Python 在这一方面有一种柔性，并非立下了军令状。

下面我们通过修改 Python 源代码来观察 Python 虚拟机进行线程调度的情形。在 ceval.c 中，我们声明一个全局的整形变量 counter，用于记录实际执行的字节码指令。在 PyEval_EvalFrameEx 的 dispatch_opcode 标记位置之后，添加代码：++counter。注意，由于 Python 虚拟机在执行一些字节码，尤其是涉及条件判断的字节码时，会发生直接跳跃的动作，所以这里 counter 记录的也并非真实的字节码数量，只是最接近真实字节码数量的一个值。在 PyEval_EvalFrameEx 的判断“_Py_Ticker < 0”成立之后，输出 counter 的值，并将 counter 重新归 0。为了观察标准调度，我们先将 thread1.py 中的两条 time.sleep 语句注释掉，thread1.py 的运行结果如图 15-9 所示。

可以看到，在标准调度下，在两个切换线程的输出语句之间，只有一个线程的输出结果，要么是主线程的输出结果，要么是子线程的输出结果。这充分说明了，标准调度是让一个线程充分执行，直到激发 Python 的模拟时钟中断（_Py_Ticker < 0），另一个线程才能有机会执行。同时，在输出结果中，我们发现每次发生线程切换时，输出的 counter 的计数结果是不定的，这与前面的分析非常吻合。

```
F:\PythonBook\Src\thread>python thread1.py
......
......
Hello from main thread 2448
Hello from main thread 2448
PyEval_EvalFrameEx : switch thread after 127 vm opcodes
Hello from sub thread 3212
Hello from sub thread 3212
......
Hello from sub thread 3212
PyEval_EvalFrameEx : switch thread after 125 vm opcodes
Hello from main thread 2448
......
```

图 15-9 标准调度下的线程切换

## 15.4.2 阻塞调度

标准调度是在 Python 执行完了可执行的指令条数之后才发生的，但是在实际中，Python 需要支持另一种触发线程调度的情形，我们称之为阻塞调度。其基本思想是：在线程 A 通过某种操作，比如说等待输入，将自身阻塞后，Python 应该将等待 GIL 的线程 B 唤醒。

考虑我们的 thread1.py，在主线程和子线程中都有 `time.sleep(1)` 的调用。假如子线程调用了 `time.sleep(1)`，那么子线程将释放 GIL，挂起自身，Python 唤醒主线程。同样，在主线程中调用 `time.sleep(1)`，也将使 Python 唤醒子线程。这展示了一种情形，即程序有时候希望将自己挂起。

除了这种线程主动放弃 GIL 的情况之外，还有另一种情形，即程序不得不挂起。还是考虑我们的 thread1.py，假如在主线程的 `thread.start_new_thread(threadProc, ())` 之后我们调用 `raw_input()`，那么主线程必须等待用户的输入，这时，主线程也不得不释放 GIL，将自身挂起。

我们通过研究 `time.sleep()` 来剖析 Python 是如何实现这种阻塞调度的。在 Python 中，`time module` 在 timemodule.c 中实现，而其中的 `sleep` 操作由 `time_sleep` 函数，进而由 `floatsleep` 函数实现。

```
[timemodule.c]
static int floatsleep(double secs)
{
	double millisecs = secs * 1000.0;
	unsigned long ul_millis;

	Py_BEGIN_ALLOW_THREADS
```

```
    /* Allow sleep(0) to maintain win32 semantics, and as decreed
     * by Guido, only the main thread can be interrupted.
     */
    ul_millis = (unsigned long)millisecs;
    if (ul_millis == 0 || main_thread != PyThread_get_thread_ident())
        Sleep(ul_millis);
    Py_END_ALLOW_THREADS
    ……
    return 0;
}
```

实际上，Sleep 机制也是平台相关的，这里我们只展示了 Win32 平台下的 sleep 实现。同时，由于 Win32 平台下的 sleep 实现也比较复杂，我们关注的焦点并不是 timemodule 的实现，而是阻塞调度机制是如何实现的，所以我们只列出了子线程调用 sleep 时涉及的相关代码。我们看到，Python 实际上是通过调用 Win32 的系统 API：sleep 来实现了阻塞的机制。那么在调用 sleep 之前，子线程肯定需要将 GIL 释放。在 floatsleep 中，我们注意到在调用 Sleep 之前，有一个 Py_BEGIN_ALLOW_THREADS，与之对应的，在调用 Sleep 之后，还有一个 Py_END_ALLOW_THREADS，正是这两个宏完成了触发 Python 进行线程调度的工作。

```
[ceval.h]
#define Py_BEGIN_ALLOW_THREADS { \
            PyThreadState *_save; \
            _save = PyEval_SaveThread();
#define Py_END_ALLOW_THREADS    PyEval_RestoreThread(_save); \
        }

[ceval.c]
PyThreadState* PyEval_SaveThread(void)
{
    PyThreadState *tstate = PyThreadState_Swap(NULL);
    if (interpreter_lock)
        PyThread_release_lock(interpreter_lock);
    return tstate;
}

void PyEval_RestoreThread(PyThreadState *tstate)
{
    if (interpreter_lock) {
        int err = errno;
        PyThread_acquire_lock(interpreter_lock, 1);
        errno = err;
    }
    PyThreadState_Swap(tstate);
}
```

在 Py_BEGIN_ALLOW_THREADS 这个宏定义的代码中，子线程释放了 GIL，这将唤醒等待 GIL 的主线程；而在 Py_END_ALLOW_THREADS 宏所定义的代码中，子线程重新申请 GIL。注意，在子线程调用了 Py_BEGIN_ALLOW_THREAD 之后，它就不再受 GIL 的约束。从这时开始，Python 的两个线程都可能被操作系统的线程调度机制选中，直到子线程通过

Py_END_ALLOW_THREADS 申请 GIL 为止，Python 又恢复为只能有一个线程被操作系统的线程调度机制选中。

这意味着 Python 的线程在某种情况下可以脱离 GIL 的控制，然而我们看到，在 Py_BEGIN_ALLOW_THREAD 和 Py_END_ALLOW_THREADS 之间，子线程并没有调用任何 Python 的 C API，只是调用了操作系统的 API，这不会导致共享资源的访问冲突，所以依然是线程安全的。开始的时候我们就说过，在理论上，Python 并不是一定要 GIL 这样的解释器级的互斥线程的机制，只要能保护共享资源即可，而当前 Python 采用的 GIL 只是多种线程互斥机制中的一种而已。

同样，对于 raw_input 而言，其最终将由 PyOS_Readline 实现，我们最终也会在 PyOS_Readline 中发现 Py_BEGIN_ALLOW_THREAD 和 Py_END_ALLOW_THREADS 联袂的身影。Python 正是通过这两个宏实现了阻塞调度机制。

有趣的是，在线程通过阻塞调度切换时，Python 内部的那个_Py_Ticker 依然会被保持，并不会被重置为 100，只有标准调度才会重置这个 Python 的模拟时钟。在图 15-10 中，清晰地显示了这一结果。注意，这时需要将 thread1.py 中的两条 time.sleep 语句打开，以便激发阻塞调度。

```
F:\PythonBook\Src\thread>python thread1.py
......
Hello from main thread 4012
Hello from sub thread 4000
PyEval_EvalFrameEx : switch thread after 139 vm opcodes
Hello from main thread 4012
Hello from sub thread 4000
Hello from sub thread 4000
......
Hello from main thread 4012
PyEval_EvalFrameEx : switch thread after 137 vm opcodes
Hello from main thread 4012
Hello from sub thread 4000
......
```

图 15-10 阻塞调度与标准调度结合下的线程切换

从图 15-10 中我们看到阻塞调度确实是独立于标准调度另一种线程调度机制。而阻塞调度确实没有重置_Py_Ticker，否则 Python 显示出来的值决不会是 137 这样小的值了。需要注意的，图 15-10 和图 15-9 中的输出从 120 多增长到了 130 多，是因为图 15-9 对应的例子中我们只使用了标准调度，图 15-10 中则混合使用了标准调度和阻塞调度，这多出来的 10 几条正是 Python 执行 time.sleep()时所消耗的指令数。

# 15.5 Python 子线程的销毁

在线程的全部计算完成之后，Python 将销毁线程。需要注意的是，Python 主线程的销毁与子线程的销毁是不同的，因为主线程的销毁动作必须要销毁 Python 的运行时环境，而子线程的销毁则不需要进行这些动作。在本节中，我们只剖析 Python 子线程的销毁过程。

通过前面的分析我们知道，线程的主体框架是在 t_bootstrap 中的：

```
[threadmodule.c]
static void t_bootstrap(void *boot_raw)
{
    struct bootstate *boot = (struct bootstate *) boot_raw;
    PyThreadState *tstate;
    PyObject *res;

    tstate = PyThreadState_New(boot->interp);
    PyEval_AcquireThread(tstate);
    res = PyEval_CallObjectWithKeywords(boot->func, boot->args, boot->keyw);
    PyMem_DEL(boot_raw);
    PyThreadState_Clear(tstate);
    PyThreadState_DeleteCurrent();
    PyThread_exit_thread();
}
```

Python 首先会通过 PyThreadState_Clear 清理当前线程所对应的线程状态对象。所谓清理，实际上比较简单，就是对线程状态对象中维护的东西进行引用计数的维护。随后，Python 释放 GIL，释放 GIL 的操作是在 PyThreadState_DeleteCurrent 中完成的。

```
[pystate.c]
void PyThreadState_DeleteCurrent()
{
    PyThreadState *tstate = _PyThreadState_Current;
    _PyThreadState_Current = NULL;
    tstate_delete_common(tstate);
    if (autoTLSkey && PyThread_get_key_value(autoTLSkey) == tstate)
        PyThread_delete_key_value(autoTLSkey);
    PyEval_ReleaseLock();
}
```

在 PyThreadState_DeleteCurrent 中，首先会删除当前的线程状态对象，然后通过 PyEval_ReleaseLock 释放 GIL。

Python 在函数 PyThreadState_DeleteCurrent 完成了绝大部分线程的销毁动作，剩下的 PyThread_exit_thread 是一个平台相关的操作，完成各个平台上不同的销毁原生线程的工作。在 Win32 下，实际上就是调用_endthread。

# 15.6 Python 线程的用户级互斥与同步

## 15.6.1 用户级互斥与同步

我们知道，Python 的线程在 GIL 的控制之下，线程之间，对整个 Python 解释器，对 Python 提供的 C API 的访问，都是互斥的，这可以看作是 Python 内核级的互斥机制。但是这种互斥是我们不能控制的，我们还需要另一种可控的互斥机制——用户级互斥。内核级通过 GIL 实现的互斥保护了内核的共享资源，同样，用户级互斥保护了用户程序中的共享资源。考虑下面的例子：

```
[thread2.py]
import thread
import time

input = None
lock = thread.allocate_lock()

def threadProc():
    while True:
        print 'sub thread id : ', thread.get_ident()
        print 'sub thread %d wait lock...' % thread.get_ident()
        lock.acquire()
        print 'sub thread %d get lock...' % thread.get_ident()
        print 'sub thread %d receive input : %s' % (thread.get_ident(), input)
        print 'sub thread %d release lock...' % thread.get_ident()
        lock.release()
        time.sleep(1)


thread.start_new_thread(threadProc, ())
print 'main thread id : ', thread.get_ident()
while True:
    print 'main thread %d wait lock...' % thread.get_ident()
    lock.acquire()
    print 'main thread %d get lock...' % thread.get_ident()
    input = raw_input()
    print 'main thread %d release lock...' % thread.get_ident()
    lock.release()
    time.sleep(1)
```

在 thread2.py 中，有一个主线程和子线程之间共享的变量 `input`。这个 `input` 是用户的输入，主线程接收输入，而子线程打印用户输入，为了保证子线程在用户输入之后才激活打印动作，thread2.py 使用了 Python 线程机制提供的 Lock 机制来实现同步动作，这实际上也可以视为线程之间的互斥。

当主线程通过 `lock.acquire` 获得 lock 之后，将独享对 `input` 的访问权利。子线程会因为等待 lock 而将自身挂起，直到主线程释放 lock 之后才会被 Python 的线程调度机制唤醒，获得访问 `input` 的权力。注意，这里主线程需要使用 `sleep` 使自身挂起，才能触发 Python 的线程调度，使得子线程获得运行的机会。而这时主线程由于等待 lock，同样会将自身挂起，不能再访问 `input`。于是，自始至终，每一个线程都能控制自己对 `input` 的使用，不用担心别的线程破坏 `input` 的状态。这种机制给了用户控制线程之间交互的能力，是 Python 中实现线程互斥和同步的核心。

在本节中，我们将详细剖析 Python 中 Lock 机制的实现，有了前面关于 Python 中线程实现的基础，你会发现，Lock 机制的实现真的可以用顺其自然来形容。在进入 Lock 机制的实现之前，我们先来看看 thread2.py 的输出结果，以对 Lock 机制有一个感性的认识。如图 15-11 所示。

```
F:\PythonBook\Src\thread>python thread2.py
......
main thread 1384 wait lock ...
main thread 1384 get lock ...
sub thread 3724 wait lock ...
Python is the best!
main thread 1384 release lock ...
sub thread 3724 get lock ...
sub thread 3724 receive input : Python is the best!
sub thread 3724 release lock ...
sub thread 3724 wait lock ...
main thread 1384 wait lock ...
main thread 1384 get lock ...
```

图 15-11 thread2.py 的运行结果

## 15.6.2 Lock 对象

在 thread2.py 中，我们通过 `thread.allocate()` 创建了一个 `Lock` 对象，所以我们对 `Lock` 对象的剖析就从 `thead.allocate` 所对应的 C 函数 `thread_PyThread_allocate_lock` 开始。

```
[threadmodule.c]
static PyObject* thread_PyThread_allocate_lock(PyObject *self)
{
    return (PyObject *) newlockobject();
}

static lockobject* newlockobject(void)
{
    lockobject *self;
    self = PyObject_New(lockobject, &Locktype);
    self->lock_lock = PyThread_allocate_lock();
    return self;
}
```

实际上，对 thread.allocate 的调用仅仅通过 newlockobject 创建了一个 lockobject 对象，Python 的整个用户级线程同步机制就在这个对象的基础上实现。

```
[pythread.h]
typedef void *PyThread_type_lock;

[threadmodule.c]
typedef struct {
    PyObject_HEAD
    PyThread_type_lock lock_lock;
} lockobject;
```

显然，lockobject 是一个 PyObject 对象，其中的 lock_lock 的类型我们在 GIL 的身上也曾见过。在 newlockobject() 中，我们清晰地看到，lock_lock 也是由 PyThread_allocate_lock 创建的，与 GIL 一样，lock_lock 也是一个 Win32 下的 Event 内核对象。这就意味着，在 Win32 平台下的 Python 实现中，其用户级线程的互斥与同步机制是通过 Event 来完成的。我们来看一看 Lock 对象所提供的属性集。

```
[threadmodule.c]
static PyMethodDef lock_methods[] = {
    {"acquire_lock",  (PyCFunction)lock_PyThread_acquire_lock, … }
    {"acquire",       (PyCFunction)lock_PyThread_acquire_lock, … }
    {"release_lock",  (PyCFunction)lock_PyThread_release_lock, … }
    {"release",       (PyCFunction)lock_PyThread_release_lock, … }
    {"locked_lock",   (PyCFunction)lock_locked_lock, … }
    {"locked",        (PyCFunction)lock_locked_lock, … }
    {NULL,          NULL}      /* sentinel */
};
```

很简单，实际上 Lock 对象仅仅提供了三种操作：acquire、release 和判断当前 Lock 是否被锁的 locked。

一个 Python 线程在内核级需要访问 Python 解释器之前，需要先申请 GIL；同样地，线程在用户级需要访问共享资源之前也需要先申请用户级的 lock，这个申请动作在 lock.acquire 中完成，其对应的 C 函数为 lock_PyThread_acquire_lock。

```
[threadmodule.c]
static PyObject *
```

```
lock_PyThread_acquire_lock(lockobject *self, PyObject *args)
{
    //i 中保存用户传入的参数，表示是否在 lock 资源不可用时将自身挂起
    //进行等待
    int i = 1;
    PyArg_ParseTuple(args, "|i:acquire", &i);

    Py_BEGIN_ALLOW_THREADS
    i = PyThread_acquire_lock(self->lock_lock, i);
    Py_END_ALLOW_THREADS
}
```

由于线程需要等待另一个 lock 资源，为了避免死锁，需要将 GIL 转交给其他的等待 GIL 的 Python 线程，调用 `lock.acquire` 的线程使用了我们之前提到的 `Py_BEGIN_ALLOW_THREADS` 来释放 GIL，唤醒其他线程，然后调用 `PyThread_acquire_lock` 开始尝试申请用户级 lock。在获得了用户级 lock 之后，通过 `Py_BEGIN_ALLOW_THREADS` 再次获得内核级 lock——GIL。

现在，我们可以轻易地猜出 lock 的 `release` 操作 `lock_PyThread_release_lock` 完成了什么操作了。

```
[threadmodule.c]
static PyObject* lock_PyThread_release_lock(lockobject *self)
{
    /* Sanity check: the lock must be locked */
    if (PyThread_acquire_lock(self->lock_lock, 0)) {
        PyThread_release_lock(self->lock_lock);
        PyErr_SetString(ThreadError, "release unlocked lock");
        return NULL;
    }

    PyThread_release_lock(self->lock_lock);
    Py_INCREF(Py_None);
    return Py_None;
}
```

## 15.7 高级线程库——threading

Python 中的 `thread module`，以及 `Lock` 对象是 Python 提供的低级的线程控制工具，为了简化多线程应用的开发，Python 在 thread 的基础上构建了一个高级的线程控制库——`threading`。在这一节中，我们将剖析 `threading` 的具体实现。在剖析 `threading` 的具体实现之前，我们先来看看 `threading` 是如何使用的。

```
[thread3.py]
import threading
import time

class MyThread(threading.Thread):
```

```
    def run(self):
        while True:
            print 'sub thread : ', threading._get_ident()
            time.sleep(1)

mythread = MyThread()
mythread.start()
while True:
    print 'main thread : ', threading._get_ident()
    time.sleep(1)
```

其执行结果如图 15-12 所示。

```
F:\PythonBook\Src\thread>python thread3.py
sub thread : 1452
main thread : 2528
sub thread : 1452
main thread : 2528
sub thread : 1452
main thread : 2528
main thread : 2528
......
```

图 15-12 thread3.py 的运行结果

## 15.7.1 Threading Module 概述

Python 的 threading module 是在建立在 thread module 基础之上的一个 module，在 threading module 中，暴露了许多 thread module 中的属性。比如我们在 thread3.py 中使用的 threading._get_ident 实际上就是 thread.get_ident。

```
[threading.py]
import thread

_start_new_thread = thread.start_new_thread
_allocate_lock = thread.allocate_lock
_get_ident = thread.get_ident
ThreadError = thread.error
```

在 threading module 中，有一套记录当前所有通过继承 threading.Thread 而创建的 Python 线程的机制。这个机制通过两个 dict 和一个 lock 完成。

```
[threading.py]
# Active thread administration
_active_limbo_lock = _allocate_lock()
_active = {}
_limbo = {}
```

我们知道通过 threading.Thread 创建多线程，有两个阶段，第一阶段是调用 threading.Thread.start，而第二阶段是在 threading.Thread.start 中调用 threading.Thread.run。当处于第一阶段时，还没有调用 thread.start_new_thread 创建原生子线程，这时候线程记录在_limbo 中。由于没有创建子线程，所以现在没有线程 id，记录的方式为_limbo[thread] = thread。在第二阶段，已经成功地调用 thread.start_new_thread 创建了原生子线程，这时将从_limbo 中删除子线程，而将子线程记录到_active 中，记录的方式为_active[thread_id] = thread。可见，Python 这两个 dict 分别维护了已经创建和等待创建的子线程集合。对这两个 dict 的访问都在_active_limbo_lock 的保护之下进行。

在 threading module 中，提供了列举当前所有子线程的操作：threading.enumerate。这个操作很简单，就是将_active 和_limbo 中维护的线程集合的信息输出。

```
[threading.py]
def enumerate():
    _active_limbo_lock.acquire()
    active = _active.values() + _limbo.values()
    _active_limbo_lock.release()
    return active
```

## 15.7.2 Threading 的线程同步工具

在 thread module 中，Python 提供了用户级的线程同步工具：Lock 对象。而在 threading module 中，Python 提供了不同的用于线程同步的工具，以简化 Python 用户实现多线程应用程序。这些 threading 中的线程同步工具实际上都是建立在 thread 所提供的 Lock 对象的基础上的。

在 threading 中，我们可以直接创建 thread 中的 Lock 对象，threading 没有做任何包装，仅仅是简单地将其展示出来。

```
[threding.py]
_allocate_lock = thread.allocate_lock
Lock = _allocate_lock
```

通过调用 threading.Lock，我们就可以创建一个 thread 中的 Lock 对象，如前面所描述的，在这个对象上，我们可以进行 acquire、release 等操作。在 threading 中的其他线程同步工具都是在这个 Lock 对象的基础上，下面我们将对这些线程同步工具做一个概述性的介绍，具体的实现请读者参阅 threading.py。

### RLock

RLock 对象是 Lock 对象的一个变种，其内部维护着一个 Lock 对象，但是它是一种可重入的 Lock。一般地，对于 Lock 对象而言，如果一个线程连续两次进行 acquire 操作，

那么由于第一次 acquire 之后没有 release，第二次 acquire 将挂起线程，这将直接导致 Lock 对象永远不会 release，因此线程死锁。RLock 对象允许一个线程多次对其进行 acquire 操作，因为在其内部通过一个 counter 变量维护着线程 acquire 的次数。而且每一次的 acquire 操作必须有一个 release 操作与之对应，在所有的 release 操作都完成之后，别的线程才能申请该 RLock 对象。

#### Condition

Condition 对象是对 Lock 对象的包装，在创建 Condition 对象时，其构造函数需要一个 Lock 对象作为参数，如果没有这个 Lock 对象参数，Condition 将在内部自行创建一个 Rlock 对象。在 Condition 对象上，当然也可以调用 acquire 和 release 操作，因为内部的 Lock 对象本身就支持这些操作。但是 Condition 的价值在于其提供的 wait 和 notify 的语义。假设有 Condition 对象 C，当线程 A 调用 C.wait()时，线程 A 将释放 C 中的 Lock 对象，并进入阻塞状态，直到有别的线程调用 C.notify()，A 才会重新通过 acquire 申请 C 中的 Lock 对象，并退出 wait 操作。

#### Semaphore

Semaphore 对象内部维护着一个 Condition 对象，对于管理一组共享资源非常有用。Lock 对象可以保护一个共享资源，但是假如我们有一个共享资源池，其中有 5 个共享资源 A，这意味着可以有 5 个线程同时自由地访问这些资源，然而如果使用 Lock 来对共享资源进行保护的话，所有的线程都将互斥，这使得有 4 个资源 A 被浪费了。Semaphore 正是在 Condition 的基础上实现的对共享资源池进行保护的线程同步机制。Semaphore 提供了两个操作：acquire 和 release，都具有与 Lock 相同的语义。当线程调用 Semaphore.acquire 时，如果共享资源池中还有剩余的 A 时，线程就会继续执行；而如果资源池中已经没有任何资源存在了，线程就会将自身挂起，直到别的线程调用 Semaphore.release 释放一个资源。

#### Event

与 Semaphore 类似，Event 对象实际上也是对 Condition 对象的一种包装，只是提供了独有的 set 和 wait 语义。Event 类的代码很简单，有兴趣的读者可以参考 threading.py。

## 15.7.3 Threading 中的 Thread

在 thread3.py 中我们看到，threading 中一个关键的组件是 threading.Thread，在这一节中我们来看一看它的具体实现。在 threading.Thread 的实现中，你会发现我们前面提到的许多机制。

```
[threading.py]
class Thread(_Verbose):
```

```
__initialized = False
def __init__(self, group=None, target=None, name=None,
             args=(), kwargs={}, verbose=None):
    ......
    self.__name = str(name or _newname())
    self.__started = False
    self.__stopped = False
    self.__block = Condition(Lock())
    self.__initialized = True

def start(self):
    _active_limbo_lock.acquire()
    _limbo[self] = self
    _active_limbo_lock.release()
    _start_new_thread(self.__bootstrap, ())
    self.__started = True
    _sleep(0.000001)    # 1 usec, to let the thread run (Solaris hack)

def run(self):
    if self.__target:
        self.__target(*self.__args, **self.__kwargs)

def __bootstrap(self):
    try:
        self.__started = True
        _active_limbo_lock.acquire()
        _active[_get_ident()] = self
        del _limbo[self]
        _active_limbo_lock.release()
        try:
            self.run()
    finally:
        self.__stop()
        try:
            self.__delete()
        except:
            pass

def __stop(self):
    self.__block.acquire()
    self.__stopped = True
    self.__block.notifyAll()
    self.__block.release()

def join(self, timeout=None):
    self.__block.acquire()
    if timeout is None:
        while not self.__stopped:
            self.__block.wait()
    else:
        deadline = _time() + timeout
        while not self.__stopped:
            delay = deadline - _time()
            if delay <= 0:
                if __debug__:
```

```
                self._note("%s.join(): timed out", self)
                break
            self.__block.wait(delay)
        else:
            if __debug__:
                self._note("%s.join(): thread stopped", self)
    self.__block.release()
```

我们看到，在调用 threading.Thread.start 时，会在_limbo 中记录线程，然后通过 thread.start_new_thread 创建原生线程，线程过程为__bootstrap。在__bootstrap 中，会从_limbo 中删除线程记录，转而将线程记录到_active 中。然后调用 run，通常用户从 threading.Thread 派生的 class 都会覆盖 run 函数，这就实现了用户自定义的线程过程。

在 threading.Thread 中，维护着一个 Condition 对象__block，在 run 结束后，会调用__stop 操作。在这个操作里会调用 Condition 对象的 notifyAll 函数通知所有等待该对象的线程。那么有哪些线程会等待这个对象呢，凡是希望等待该线程结束消息的线程，都会通过 threading.Thread.join 操作注册成为 Condition 对象的等待线程。可以看到，在 join 中，调用了 Condition 对象的 wait 操作。在这里，我们看到了 Observer Pattern 的影子，与我们通常在 OO 语言中所见的 Observer Pattern 的应用绝然不同，但是毫无疑问，这的的确确闪现着 Observer Pattern 的身影。

对于 threading.Thread，我们的介绍就到此为止，更加细节的东西，请参与 threading.py 文件。

CHAPTER

第 16 章

# Python 的内存管理机制

内存管理，对于 Python 这样的动态语言，是至关重要的一部分，它在很大程度上甚至决定了 Python 的执行效率，因为在 Python 的运行中，会创建和销毁大量的对象，这些都涉及内存的管理。另一方面，和 Java、C#这些编程语言一样，Python 提供了对内存的垃圾收集（GC）机制，将开发者从繁琐的手动维护内存的工作中解放出来。Python 中的 GC 机制又是如何实现的呢？在这一章中，我们将会细致地剖析 Python 内部所采用的内存管理机制。

## 16.1 内存管理架构

在剖析 Python 的内存管理架构之前，有一点需要说明。Python 中所有的内存管理机制都有两套实现，这两套实现由编译符号 `PYMALLOC_DEBUG` 控制，当该符号被定义时，使用的是 debug 模式下的内存管理机制，这套机制在正常的内存管理动作之外，还会记录许多关于内存的信息，以方便 Python 在开发时进行调试；而当该符号未被定义时，Python 的内存管理机制只进行正常的内存管理动作。在本章中，我们将关注的焦点只放在非 debug 模式下的内存管理机制上。

在 Python 中，内存管理机制被抽象成一种层次似的结果，如图 16-1 所示。

在最底层（第 0 层），是操作系统提供的内存管理接口，比如 C 运行时所提供的 `malloc` 和 `free` 接口。这一层是由操作系统实现并管理的，Python 不能干涉这一层的行为。从这一层往上，剩余的三层都是由 Python 实现并维护的。

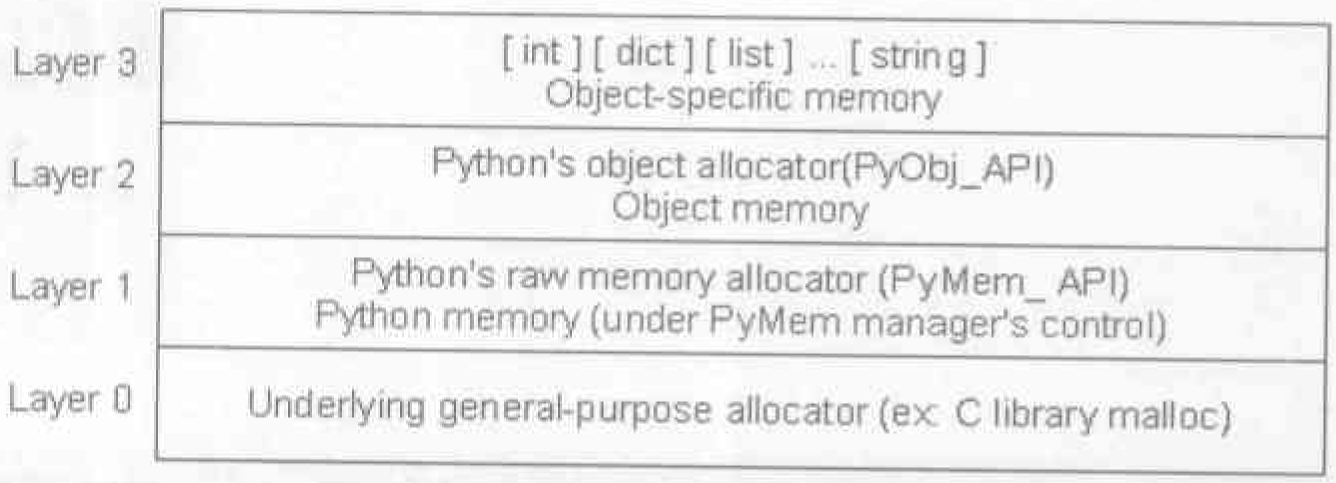



图 16-1　Python 内存管理机制的层次结构

第一层是 Python 基于第 0 层操作系统的内存管理接口包装而成的，这一层并没有在第 0 层上加入太多的动作，其目的仅仅是为 Python 提供一层统一的 raw memory 的管理接口。Python 是用 C 实现的，为什么还需要在 C 所提供的内存管理接口之上再提供一层并不太多实际意义的包装层呢？这是因为虽然不同的操作系统都提供了 ANSI C 标准所定义的内存管理接口，但是对于某些特殊的情况不同操作系统有不同的行为。比如调用 `malloc(0)`，有的操作系统会返回 `NULL`，表示内存申请失败；然而有的操作系统会返回一个貌似正常的指针，但是这个指针所指的内存并不是有效的。为了最广泛的可移植性，Python 必须保证相同的语义一定代表着相同的运行时行为，为了处理这些与平台相关的内存分配行为，Python 必须要在 C 的内存分配接口之上再提供一层包装。

在 Python 中，第一层的实现就是一组以 `PyMem_` 为前缀的函数族，下面我们来看一看这第一层内存管理机制：

```
[pymem.h]
PyAPI_FUNC(void *) PyMem_Malloc(size_t);
PyAPI_FUNC(void *) PyMem_Realloc(void *, size_t);
PyAPI_FUNC(void) PyMem_Free(void *);

[object.c]
void* PyMem_Malloc(size_t nbytes)
{
    return PyMem_MALLOC(nbytes);
}

void* PyMem_Realloc(void *p, size_t nbytes)
{
    return PyMem_REALLOC(p, nbytes);
}

void PyMem_Free(void *p)
{
    PyMem_FREE(p);
}
```

我们看到，在第一层，Python 提供了类似于 C 中 `malloc`、`realloc`、`free` 的语义。

有趣的是，PyMem_Malloc 是通过一个宏 PyMem_MALLOC 来实现的，对于 PyMem_Realloc 和 PyMem_Free，情况也是如此：

```
[pymem.h]
#define PyMem_MALLOC(n)          malloc((n) ? (n) : 1)
#define PyMem_REALLOC(p, n)      realloc((p), (n) ? (n) : 1)
#define PyMem_FREE               free
```

显然，PyMem_Malloc 其实就是 C 中的 malloc，只是对申请大小为 0 的内存这种特殊情况进行了处理。Python 不允许申请大小为 0 的内存空间，它会强制将其装换为申请大小为 1 个字节的内存空间，从而避免了不同操作系统上的不同运行时的行为。

这里有一个有趣的问题，为什么 Python 会同时提供函数和宏这两套接口呢？其实，这正是 Python 为了执行效率殚精竭虑的表现。使用宏可以避免一次函数调用的开销，提高运行效率。但是对于用户使用 C 来编写 Python 的扩展模块时，使用宏是危险的，因为随着 Python 的不断演进，其内存管理机制的具体实现很可能会发生变化，所以虽然 Python 中宏的名字是不会变的，但是其所代表的实现代码是会变的。因此，使用旧的宏编写的 C 扩展模块就可能与新版本的 Python 产生二进制不兼容，这是一个非常危险的陷阱。因此 Python 对于这些内存管理接口，总是同时提供函数和宏这两套接口，这一点我们在后面会经常看到。如果使用 C 来编写 Python 的扩展模块，使用函数接口是一个良好的编程习惯。

到现在为止，我们所介绍的内存管理接口都与 malloc 等有相同的语义，仅仅是分配 raw memory 而已。其实在第一层中，Python 还提供了面向 Python 中类型的内存分配器：

```
[pymem.h]
#define PyMem_New(type, n) \
  ( (type *) PyMem_Malloc((n) * sizeof(type)) )
#define PyMem_NEW(type, n) \
  ( (type *) PyMem_MALLOC((n) * sizeof(type)) )

#define PyMem_Resize(p, type, n) \
  ( (p) = (type *) PyMem_Realloc((p), (n) * sizeof(type)) )
#define PyMem_RESIZE(p, type, n) \
  ( (p) = (type *) PyMem_REALLOC((p), (n) * sizeof(type)) )

#define PyMem_Del        PyMem_Free
#define PyMem_DEL        PyMem_FREE
```

在 PyMem_Malloc 中，和 malloc 一样，程序员需要自行提供所申请空间的大小。然而在 PyMem_New 中，只需要提供类型和数量，Python 会自动侦测其所需的内存空间大小。

第一层所提供的内存管理接口其功能是有限的，想象一下，假如我要创建一个 PyIntObject 对象，还需要进行许多额外的工作，比如设置对象的类型对象参数，初始化对象的引用计数值等。为了简化 Python 自身的开发，Python 在比第一层更高的抽象层次上提供了第二层内存管理接口。在这一层，是一组以 PyObje_为前缀的函数族，主要提供了创建 Python 对象的接口。这一套函数族又被唤作 Pymalloc 机制，从 Python2.1 开始，

它才慢慢登上历史舞台，在 Python 2.1 和 Python 2.2 中，这个机制是作为实验性质的机制，所以默认情况下是不打开的，只有自己通过——with-pymalloc 编译符号重新编译 Python，才能激活 Pymalloc 机制。在 Python 2.3 发布时，Pymalloc 机制已经经过了长期的优化和稳定，终于登上了“正宫”的位置，在默认情况下被打开了。

在第二层内存管理机制之上，对于 Python 中的一些常用对象，比如整数对象、字符串对象等，Python 又构建了更高抽象层次的内存管理策略。对于第三层的内存管理策略，主要就是对象缓冲池机制，这一点在本书第一部分我们已经剖析了。而第一层的内存管理机制仅仅是对 malloc 的简单包装，真正在 Python 中发挥巨大作用、同时也是 GC 的藏身之处的内存管理机制，就在第二层内存管理机制中。所以，从下面开始，我们将进入对这套机制的剖析。

## 16.2 小块空间的内存池

在 Python 中，许多时候申请的内存都是小块的内存，这些小块内存在申请后，很快又会被释放，由于这些内存的申请并不是为了创建对象，所以并没有对象一级的内存池机制。这就意味着 Python 在运行期间会大量地执行 malloc 和 free 的操作，导致操作系统频繁地在用户态和核心态之间进行切换，这将严重影响 Python 的执行效率。为了提高 Python 的执行效率，Python 引入了一个内存池机制，用于管理对小块内存的申请和释放。这也就是之前提到的 Pymalloc 机制。前面我们已经看到，这套机制在 Python 2.5 中默认是启动了的，也就是说，在 Python 2.5 中，用于管理小块内存的内存池就被激活了，并通过 PyObject_Malloc、PyObject_Realloc 和 PyObject_Free 三个接口显示给 Python。

在 Python 2.5 中，整个小块内存的内存池可以视为一个层次结构，在这个层次结构中，一共分为 4 层，从下至上分别是：block、pool、arena 和内存池。需要说明的是，block、pool 和 arena 都是 Python 代码中可以找到的实体，而最顶层的“内存池”只是一个概念上的东西，表示 Python 对于整个小块内存分配和释放行为的内存管理机制。

### 16.2.1 Block

在最底层，block 是一个确定大小的内存块。在 Python 中，有很多种 block，不同种类的 block 都有不同的内存大小，这个内存大小的值被称为 size class。为了在当前主流的 32 位平台和 64 位平台上都能获得最佳的性能，所有的 block 的长度都是 8 字节对齐的。

```
[obmalloc.c]
#define ALIGNMENT       8       /* must be 2^N */
#define ALIGNMENT_SHIFT     3
#define ALIGNMENT_MASK      (ALIGNMENT - 1)
```

同时，Python 为 block 的大小设定了一个上限，当申请的内存大小小于这个上限时，Python 可以使用不同种类的 block 来满足对内存的需求；当申请的内存大小超过了这个上限，Python 就会将对内存的请求转交给第一层的内存管理机制，即 PyMem 函数族，来处理。这个上限值在 Python 2.5 中被设置为 256。

```
[obmalloc.c]
#define SMALL_REQUEST_THRESHOLD    256
#define NB_SMALL_SIZE_CLASSES   (SMALL_REQUEST_THRESHOLD / ALIGNMENT)
```

根据 SMALL_REQUEST_THRESHOLD 和 ALIGNMENT 的限定，实际上，我们可以由此得到不同种类的 block 的 size class 分别为：8，16，32，…，256。每个 size class 对应一个 size class index，这个 index 从 0 开始。所以对于小于 256 字节的小块内存的分配，我们可以得到如下的结论：

```
* Request in bytes     Size of allocated block    Size class idx
* ----------------------------------------------------------------
*    1-8                    8                        0
*    9-16                  16                        1
*   17-24                  24                        2
*   25-32                  32                        3
*   33-40                  40                        4
*   41-48                  48                        5
*   49-56                  56                        6
*   57-64                  64                        7
*   65-72                  72                        8
*    ...                   ...                      ...
*  241-248                248                       30
*  249-256                256                       31
*
*  0, 257 and up: routed to the underlying allocator.
```

也就是说，当我们申请一块大小为 28 字节的内存时，实际上 PyObject_Malloc 从内存池中划给我们的内存是 32 字节的一个 block，从 size class index 为 3 的 pool（关于 pool 到底是什么劳什子，参见下一节的剖析，这里可以暂且想象成 block 的集合）中划出。下面的两个式子给出了在 size class 和 size class index 之间的转换：

```
[obmalloc.c]
//从size class index转换到size class
#define INDEX2SIZE(I) (((uint)(I) + 1) << ALIGNMENT_SHIFT)

//size class转换到size class index
size = (uint )(nbytes - 1) >> ALIGNMENT_SHIFT;
```

现在，需要指出一个相当关键的点，虽然我们这里谈论了很多 block，但是在 Python 中，block 只是一个概念，在 Python 源码中没有与之对应的实体存在。之前我们说对象，对象在 Python 源码中有对应的 PyObject；我们说列表，列表在 Python 源码中对应 PyListObject、PyType_List。这里的 block 就很奇怪了，它仅仅是概念上的东西，我们知道它是具有一定大小的内存，但它不与 Python 源码里的某个东西对应。然而，Python 却提供了一个管理 block 的东西，这就是下面要剖析的 pool。

## 16.2.2 Pool

一组 block 的集合称为一个 pool，换句话说，一个 pool 管理着一堆有固定大小的内存块。事实上，pool 管理着一大块内存，它有一定的策略，将这块大的内存划分为多个小的内存块。在 Python 中，一个 pool 的大小通常为一个系统内存页，由于当前大多数 Python 支持的系统的内存页都是 4KB，所以 Python 内部也将一个 pool 的大小定义为 4KB。

```
[obmalloc.c]
#define SYSTEM_PAGE_SIZE     (4 * 1024)
#define SYSTEM_PAGE_SIZE_MASK   (SYSTEM_PAGE_SIZE - 1)

#define POOL_SIZE       SYSTEM_PAGE_SIZE    /* must be 2^N */
#define POOL_SIZE_MASK      SYSTEM_PAGE_SIZE_MASK
```

虽然 Python 没有为 block 提供对应的结构，但它对于 pool 却是相当照顾的。Python 源码中的 `pool_header` 就是为 pool 这个概念提供的实现。

```
[obmalloc.c]
typedef uchar block;

/* Pool for small blocks. */
struct pool_header {
    union { block *_padding;
        uint count; } ref;  /* number of allocated blocks    */
    block *freeblock;       /* pool's free list head         */
    struct pool_header *nextpool;   /* next pool of this size class  */
    struct pool_header *prevpool;   /* previous pool       ""        */
    uint arenaindex;        /* index into arenas of base adr */
    uint szidx;         /* block size class index    */
    uint nextoffset;        /* bytes to virgin block     */
    uint maxnextoffset;     /* largest valid nextoffset  */
};
```

我们刚才说了一个 pool 的大小在 Python2.5 中是 4KB，但是看看这个 `pool_header` 呢？就算是用大腿看也能看出 `pool_header` 吃不完 4KB 的内存。关键就在于“`header`”这个词，原来这个 `pool_header` 仅仅是一个 pool 的头部，4KB 的内存，除去 `pool_header`，还有很大一块。还记得我们说过 pool 管理着一堆 block 吗？对了，这剩下的很大一块的内存就是 pool 中维护的 block 的集合占用的内存。

前面提到 block 是有固定大小的内存块，因此，pool 也携带了大量这样的信息。一个 pool 管理的所有 block，它们的大小都是一样的。也就是说，一个 pool 可能管理了 100 个 32 个字节的 block，也可能管理了 100 个 64 个字节的 block，但是绝不会有一个管理了 50 个 32 字节的 block 和 50 个 64 字节的 block 的 pool 存在。每一个 pool 都和一个 size 联系在一起，更确切地说，都和一个 size class index 联系在一起。这就是 `pool_header` 中的 szindex 的意义。

假设我们手上现在有一块 4KB 的内存，来看看 Python 是如何将这块内存改造为一个

管理 32 字节 block 的 pool，并从 pool 中取出第一块 block 的。

```
[obmalloc.c]-[convert 4k raw memory to pool]
#define ROUNDUP(x)        (((x) + ALIGNMENT_MASK) & ~ALIGNMENT_MASK)
#define POOL_OVERHEAD       ROUNDUP(sizeof(struct pool_header))
#define struct pool_header* poolp
#define uchar block

poolp pool;
block* bp;
...... // pool 指向了一块 4kB 的内存
pool->ref.count = 1;
//设置 pool 的 size class index
pool->szidx = size;
//将 size class index 转换为 size，比如 3 转换为 32 字节
size = INDEX2SIZE(size);
//跳过用于 pool_header 的内存，并进行对齐
bp = (block *)pool + POOL_OVERHEAD;
//实际就是 pool->nextoffset = POOL_OVERHEAD+size+size
pool->nextoffset = POOL_OVERHEAD + (size << 1);
pool->maxnextoffset = POOL_SIZE - size;
pool->freeblock = bp + size;
*(block **)(pool->freeblock) = NULL;
return (void *)bp;
```

最后返回的 bp 就是指向从 pool 中取出的第一块 block 的指针。也就是说，pool 中第一块 block 已经被分配了，所以在 ref.count 中记录了当前已经被分配的 block 数量，这时为 1。特别需要注意的是，bp 返回的实际是一个地址，这个地址之后有将近 4KB 的内存实际上都是可用的，但是可以肯定申请内存的函数只会使用[bp, bp+size]这个区间的内存，这是由 size class index 可以保证的。好了，来看一看图 16-2 所示的一块经过改造后的 4KB 内存。

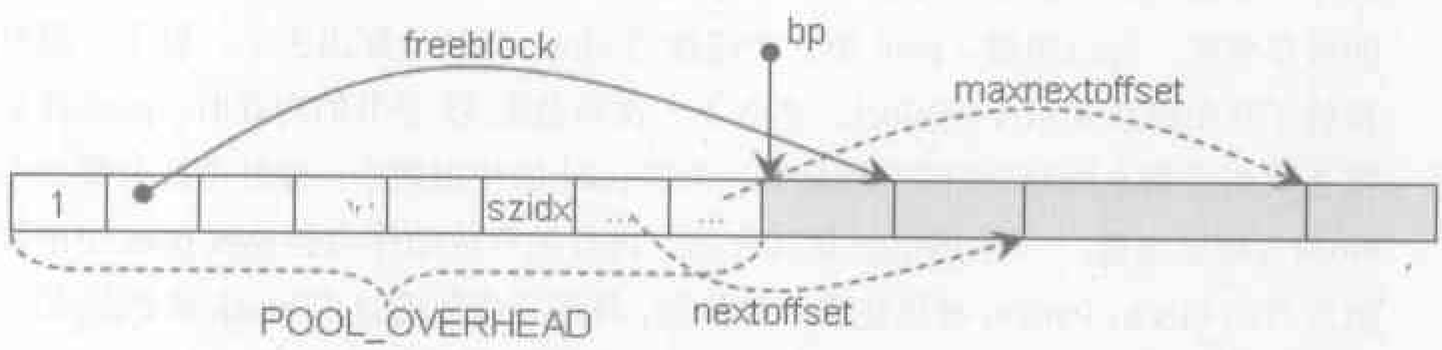



图 16-2 改造成 pool 后的 4kB 内存

注意其中的实线箭头是指针，但是虚线箭头不是代表指针，是偏移位置的形象表示。在 nextoffset 和 maxnextoffset 中存储的是相对于 poo 头部的偏移位置。

在了解了初始化后的 pool 的样子之后，可以来看看 Python 在申请 block 时，pool_header 中的各个域是怎么变动的。假设我们从现在开始连续申请 5 块 28 字节内存，由于 28 个字节对应的 size class index 为 3，所以实际上会申请 5 块 32 字节的内存。

```
[obmalloc.c]-[allocate block]
        if (pool != pool->nextpool) {
            ++pool->ref.count;
            bp = pool->freeblock;
            ......
            if (pool->nextoffset <= pool->maxnextoffset) {
                //有足够的 block 空间
                pool->freeblock = (block *)pool + pool->nextoffset;
                pool->nextoffset += INDEX2SIZE(size);
                *(block **)(pool->freeblock) = NULL;
                return (void *)bp;
            }
        }
```

原来 freeblock 指向的是下一个可用的 block 的起始地址，这一点在图 16-2 中也可以看得出。当再次申请 32 字节的 block 时，只需返回 freeblock 指向的地址就可以了，很显然，这时 freeblock 需要向前进，指向下一个可用的 block。这时，nextoffset 现身了。

在 pool header 中，nextoffset 和 maxoffset 是两个用于对 pool 中的 block 集合进行迭代的变量：从初始化 pool 的结果及图 16-2 中可以看到，它所指示的偏移位置正好指向了 freeblock 之后的下一个可用的 block 的地址。从这里分配 block 的动作也可以看到，在分配了 block 之后，freeblock 和 nextoffset 都会向前移动一个 block 的距离，如此反复，就可对所有的 block 进行一次遍历。而 maxnextoffset 指名了该 pool 中最后一个可用的 block 距 pool 开始位置的便移，它界定了 pool 的边界，当 nextoffset > maxnextoffset 时，也就意味着已经遍历完了 pool 中所有的 block 了。

嗯，申请，前进，申请，前进，这个过程非常自然，也容易理解。但是且慢，这好像意味着一个 pool 中只能满足 POOL_SIZE/size 次对 block 的申请，这很难让人接受。如果这样不容易理解，我们来考虑一个形象的例子。现在我们已经进行了 5 次连续的 32 字节的内存分配，可以想像，pool 中 5 个连续的 block 都被分配出去了。过了一段时间，程序释放了其中第 2 和第 4 块 block，那么下一次再分配 32 字节的内存时，pool 提交的应该是第 2 块还是第 6 块 block 呢？很显然，为了 pool 的使用效率，最好再次分配自由的第 2 块 block。可以想像，一旦 Python 运转起来，内存的释放动作将会导致 pool 中出现大量的离散的自由 block，Python 必须建立一种机制，将这些离散的自由 block 组织起来，再次使用。这个机制就是所谓的自由 block 链表。这个链表的关键就着落在 pool_header 中的那个 freeblock 身上。

刚才我们就说了，当 pool 初始化完成之后，freeblock 指向了一个有效的地址，为下一个可以分配出去的 block 的地址。然而奇特的是，Python 在设置了 freeblock 之后，还设置了 *freeblock。这一个动作似乎非常诡异，然而我们马上就会看到，设置 *freeblock 的动作正是建立离散自由 block 链表的关键所在。目前我们看到的 freeblock 只是在机械地前进前进，这是因为它在等待一个特殊的时刻，在这个特殊的时刻，你会发

现 freeblock 开始成为一个苏醒的精灵，在这 4KB 内存上开始灵活地舞动。这个特殊的时刻就是一个 block 被释放的时刻（见代码清单 16-1）。

**代码清单 16-1**

```
[obmalloc.c]
//基于地址 P 获得离 P 最近的 pool 的边界地址
#define POOL_ADDR(P) ((poolp)((uptr)(P) & ~(uptr)POOL_SIZE_MASK))

void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);
    //判断 p 指向的 block 是否属于 pool
    if (Py_ADDRESS_IN_RANGE(p, pool)) {
        *(block **)p = lastfree = pool->freeblock; //[1]
        pool->freeblock = (block *)p;              //[2]
        ......
    }
}
```

在释放 block 时，神秘的 freeblock 惊鸿一现，覆盖在 freeblock 身上那层神秘的面纱就要揭开了。我们知道，这时 freeblock 虽然指向了一个有效的 pool 内地址，但是 *freeblock 是为 NULL 的。假设这时 Python 释放的是 block A，在代码清单 16-1 的[1]之后，A 中第一个字节的值被设置为了当前 freeblock 的值，而在[2]之后，freeblock 的值被更新了，指向了 block A 的首地址。就是这短短的两步，一个 block 被插入到了离散自由 block 链表中。所以当第 2 块和第 4 块 block 都被释放之后，我们可以看到一个初具规模的离散自由 block 链表了，如图 16-3 所示。

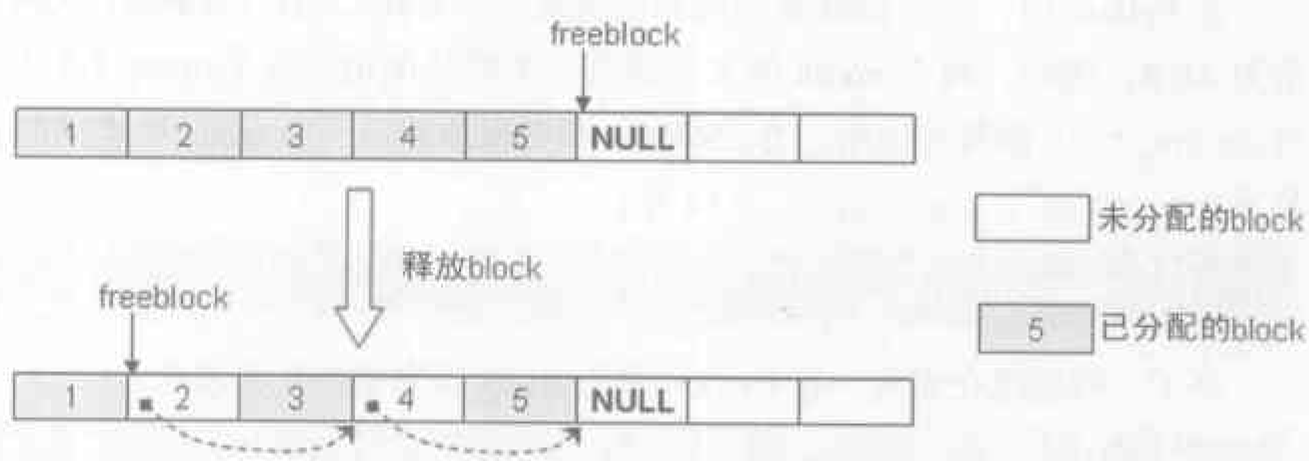



**图 16-3 释放了 block 之后产生的自由 block 链表**

到了这里，这条实现方式非常奇特的离散自由 block 链表被我们挖掘出来了。从 freeblock 开始，我们可以很容易地以 freeblock = *freeblock 的方式遍历这条链表，而当发现了*freeblock 为 NULL 时，则表明到达了该链表的尾部了。这条链表将影响到在

本小节开始时我们剖析过的一般的 block 分配行为（见代码清单 16-2）。

**代码清单 16-2**

```
[obmalloc.c]-[allocate block]
    ......
    if (pool != pool->nextpool) {
        /*
         * There is a used pool for this size class.
         * Pick up the head block of its free list.
         */
        ++pool->ref.count;
        bp = pool->freeblock; //[1]
        if ((pool->freeblock = *(block **)bp) != NULL) { //[2]
            return (void *)bp;
        }
        if (pool->nextoffset <= pool->maxnextoffset) {
            ......
        }
        ......
    }
```

这里的代码清单 16-2 的[1]和[2]正是 Python 中 `freeblock = *freeblock` 的实现方式。当[2]处的判断为真时，表明已经不存在离散自由 block 链表了，如果可能，则会继续分配 pool 的 `nextoffset` 指定的下一块 block。但是，如果连 `nextoffset <= maxnextoffset` 都不成立了呢？嗨，老兄，别忘了，我们现在谈论的仅仅是一个 pool，如果这个 pool 中的 block 被用光了，最简单的解决方案就是：我给你另一个 pool。这就意味着，在 Python 中，存在一个 pool 的集合。

## 16.2.3 arena

在 Python 中，多个 pool 聚合的结果就是一个 arena。上一节提到，pool 的大小的默认值为 4KB，同样，每个 arena 的大小都有一个默认的值。在 Python 2.5 中，这个值由名为 `ARENA_SIZE` 的符号控制，为 256KB。那么很显然，一个 arena 中容纳的 pool 的个数就是 `ARENA_SIZE / POOL_SIZE = 64` 个。

```
[obmalloc.c]
#define ARENA_SIZE      (256 << 10) /* 256KB */
```

好了，我们现在来看一看 Python 中的 arena 到底是个什么东西。

```
[obmalloc.c]
typedef uchar block;

struct arena_object {
    uptr address;
    block* pool_address;
    uint nfreepools;
    uint ntotalpools;
```

```
    struct pool_header* freepools;

    struct arena_object* nextarena;
    struct arena_object* prevarena;
};
```

一个概念上的 arena 在 Python 源码中就对应 arena_object 结构体，确切地说，arena_object 仅仅是一个 arena 的一部分，就像 pool_header 只是 pool 的一部分一样。一个完整的 arena 包括一个 arena_object 和透过这个 arena_object 管理着的 pool 集合。同样，pool 的情况类似，一个完整的 pool 包括一个 pool_header 和透过这个 pool_header 管理着的 block 集合。

#### 16.2.3.1 "未使用"的 arena 和"可用"的 arena

在 arena_object 结构体的定义中，我们看到了 nextarea 和 prevarea 这两个东西，这似乎意味着在 Python 中会有一个多个 arena 构成的链表，这个链表的表头就是 arenas。呃，这种猜测只对了一半，实际上，在 Python 中，确实会存在多个 arena_object 构成的集合，但是这个集合并不构成链表，而是构成了一个 arena 的数组。数组的首地址由 arenas 维护，这个数组就是 Python 中的通用小块内存的内存池；另一方面，nextarea 和 prevarea 也确实是用来连接 arena_object 组成链表的，既然多个 arena_object 已经通过数组组织起来了，为什么又要搞出一个链表来。乍一看，真的有点稀奇古怪。

我们曾说 arena 是用来管理一组 pool 的集合的，arena_object 的作用看上去和 pool_header 的作用是一样的，但是实际上，pool_header 管理的内存和 areana_object 管理的内存有一点细微的差别。pool_header 管理的内存与 pool_header 自身是一块连续的内存，而 areana_object 与其管理的内存则是分离的。这种差别如图 16-4 所示。

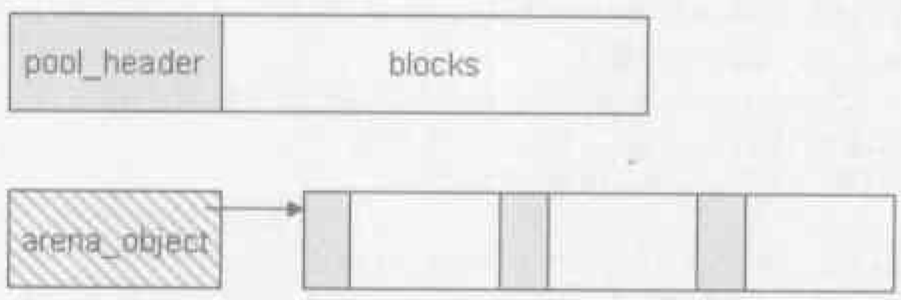



图 16-4 pool 和 arena 的内存布局区别

初看上去，这似乎没什么大不了的，不就一个是连着的，一个是分开的吗？但是这后面隐藏着这样一个事实：当 pool_header 被申请时，它所管理的 block 集合的内存一定也被申请了；但是当 aerna_object 被申请时，它所管理的 pool 集合的内存则没有被申请。换句话说，arena_object 和 pool 集合在某一时刻需要建立联系。注意，这个建立联系的时刻是一个关键的时刻，Python 从这个时刻一刀切下，将一个 arena_object 切分为两种状态。

当一个 arena 的 area_object 没有与 pool 集合建立联系时，这时的 arena 处于“未使用”状态；一旦建立了联系，这时 arena 就转换到了“可用”状态。对于每一种状态，都有一个 arena 的链表。“未使用”的 arena 的链表表头是 unused_arena_objects、arena 与 arena 之间通过 nextarena 连接，是一个单向链表；而“可用”的 arena 的链表表头是 usable_arenas、arena 与 arena 之间通过 nextarena 和 prevarena 连接，是一个双向链表。图 16-5 展示了某一时刻多个 arena 的一个可能状态。

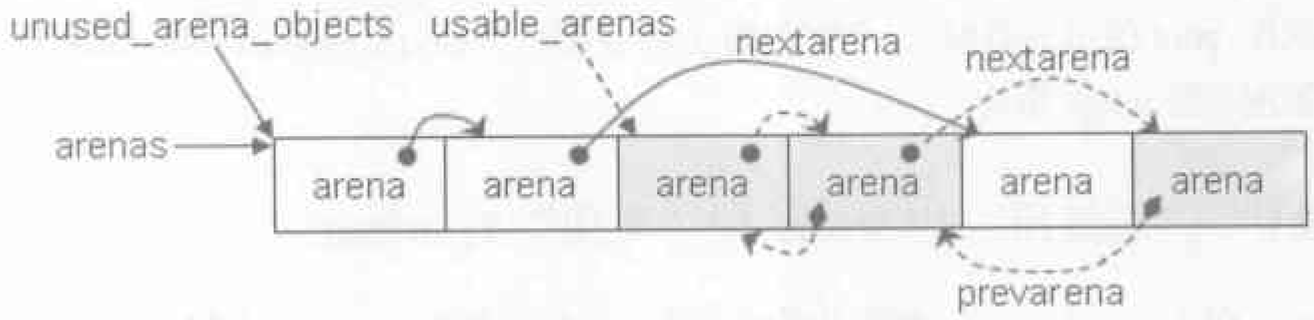



图 16-5 arena 集合在某时刻的可能状态

### 16.2.3.2 申请 arena

在运行期间，Python 使用 new_arena 来创建一个 arena，我们来看看 arena 创建时的情形（见代码清单 16-3）。

代码清单 16-3

```
[obmalloc.c]
//arenas 管理着 arena_object 的集合
static struct arena_object* arenas = NULL;
//当前 arenas 中管理的 arena_object 的个数
static uint maxarenas = 0;
//"未使用的"arena_object 链表
static struct arena_object* unused_arena_objects = NULL;
//"可用的"arena_object 链表
static struct arena_object* usable_arenas = NULL;
//初始化时需要申请的 arena_object 的个数
#define INITIAL_ARENA_OBJECTS 16

static struct arena_object* new_arena(void)
{
    struct arena_object* arenaobj;
    uint excess;    /* number of bytes above pool alignment */

    //[1]: 判断是否需要扩充“未使用的” arena_object 列表
    if (unused_arena_objects == NULL) {
        uint i;
        uint numarenas;
        size_t nbytes;

        //[2]: 确定本次需要申请的 arena_object 的个数，并申请内存
        numarenas = maxarenas ? maxarenas << 1 : INITIAL_ARENA_OBJECTS;
```

```
    if (numarenas <= maxarenas)
        return NULL; //overflow（溢出）
    nbytes = numarenas * sizeof(*arenas);
    if (nbytes / sizeof(*arenas) != numarenas)
        return NULL; //overflow
    arenaobj = (struct arena_object *)realloc(arenas, nbytes);
    if (arenaobj == NULL)
        return NULL;
    arenas = arenaobj;

    //[3]: 初始化新申请的arena_object，并将其放入unused_arena_objects链表中
    for (i = maxarenas; i < numarenas; ++i) {
        arenas[i].address = 0;    /* mark as unassociated */
        arenas[i].nextarena = i < numarenas - 1 ? &arenas[i+1] : NULL;
    }
    /* Update globals. */
    unused_arena_objects = &arenas[maxarenas];
    maxarenas = numarenas;
}

//[4]: 从unused_arena_objects链表中取出一个"未使用的"arena_object
arenaobj = unused_arena_objects;
unused_arena_objects = arenaobj->nextarena;
assert(arenaobj->address == 0);

//[5]: 申请arena_object管理的内存
arenaobj->address = (uptr)malloc(ARENA_SIZE);
++narenas_currently_allocated;

//[6]: 设置pool集合的相关信息
arenaobj->freepools = NULL;
arenaobj->pool_address = (block*)arenaobj->address;
arenaobj->nfreepools = ARENA_SIZE / POOL_SIZE;
//将pool的起始地址调整为系统页的边界
excess = (uint)(arenaobj->address & POOL_SIZE_MASK);
if (excess != 0) {
    --arenaobj->nfreepools;
    arenaobj->pool_address += POOL_SIZE - excess;
}
arenaobj->ntotalpools = arenaobj->nfreepools;

return arenaobj;
}
```

在代码清单 16-3 的[1]处，Python 首先会检查当前 `unused_arena_objects` 链表中是否还有“未使用”的 arena，检查的结果将决定后续的动作。

如果在 `unused_arena_objects` 中还存在“未使用”的 arena，那么 Python 将直接开始代码清单 16-3 的[4]处的动作，从 `unused_arena_objects` 中抽取出一个 arena，接着调整 `unused_arena_objects`，与抽取出的 arena 彻底断绝一切联系。然后，在代码清单 16-3 的[5]处，Python 申请了一块大小为 `ARENA_SIZE`（256KB）的内存，将申请的内存的地址赋给 arena 的 address。我们已经知道，arena 中维护的是 pool 的集合，这块 256KB 的内存

就是 pool 的容身之处，这时 arena_object 就和 pool 集合建立联系了，这个 arena 已经具备了成为“可用”内存的条件。到了这里，arena 和 unused_arena_objects 已经脱离了关系，就等着 usable_arenas 这个组织的接收了，到底什么时候才能接收呢，别急，谜底一会儿揭晓☺。

随后，在代码清单 16-3 的[6]处，Python 设置了一些 arena 用于维护 pool 集合的信息。特别注意的是，在[6]的动作中，Python 将申请到的 256KB 内存进行了处理，放弃了一些内存，而将可使用的内存边界（pool_address）调整到了与系统页对齐。代码清单 16-3 的[6]处将 freepools 设置为 NULL，基于前面我们对 pool 中 freeblock 的了解，这没什么大惊小怪的，看来要等到释放一个 pool 时，这个 freepools 才有用。最后我们看到，pool 集合所占用的 256KB 的内存在进行边界对齐后，实际上是交给 pool_address 来维护了。

回到 new_arena 中代码清单 16-3 的[1]处的判断，如果 unused_arena_objects 为 NULL，则表明目前系统中已经没有“未使用”的 arena 了，Python 将首先扩大系统的 arena 集合（小块内存内存池）。Python 在内部通过一个唤作 maxarenas 的变量维护了在 arenas 指向的数组中存储的 arena_object 的个数。在[2]处，Python 将待申请的 arena_object 的个数设置为当前 arena_object 个数（maxarenas）的 2 倍。当然，在首次初始化时，maxarenas 为 0，这时，Python 将新的 maxarenas 初始化为 16。

在获得了新的 maxarenas 后，Python 会检查这个新得到的值是否溢出了。如果检查顺利通过，Python 在代码清单 16-3 的[3]处通过 realloc 扩大 arenas 指向的内存，并对新申请的 arena_object 进行设置，特别要提到的是那个貌似毫不起眼的 address，[3]处将新申请的 arena 的 address 一律设置为 0。实际上，这是一个标识一个 arena 是处于“未使用”状态还是“可用”状态的重要标记。看看[6]处，一旦 arena 的 arena_object 与 pool 集合建立了联系，这个 address 就变成了非 0。当然，别忘了我们为什么会来到[3]这里的，咱们可不是饭后随便溜达到这里的，是因为那个重要的 unused_arena_objects 变为 NULL 了。对喽，所以最后还设置了 unused_arena_objects。这样一来，系统中又有了“未使用”的 arena 了，接下来，Python 就将进入[4]处对一个 arena 的初始化了。

## 16.2.4 内存池

### 16.2.4.1 可用 pool 缓冲池——usedpools

在 Python 2.5 中，Python 内部默认的小块内存与大块内存的分界点定在 256 个字节，这个分界点由前面我们看到的名为 SMALL_REQUEST_THRESHOLD 的符号控制。也就是说，当申请的内存小于 256 字节时，PyObject_Malloc 会在内存池中申请内存；当申请的内存

大于 256 字节时，PyObject_Malloc 的行为将蜕化为 malloc 的行为。当然，通过修改 Python 源代码，我们可以改变这个默认值，从而改变 Python 的默认内存管理行为。

当 Python 申请小于 256 字节的内存时，Python 会使用 arenas 所维护的内存空间。那么 Python 内部对于 arena 的个数是否有限制呢？换句话说，Python 对于这个小块空间内存池的大小是否有限制？这个决策取决于用户，Python 提供了一个编译符号，用于控制是否限制这个内存池的大小。

当 Python 在 WITH_MEMORY_LIMITS 编译符号打开的背景下进行编译时，Python 内部的另一个符号会被激活，这个名为 SMALL_MEMORY_LIMIT 的符号限制了整个内存池的大小，同时，也就限制了可以创建的 arena 的个数。在默认情况下，不论是 Win32 平台，还是 unix 平台，这个编译符号都是没有打开的，所以通常 Python 都没有对小块内存的内存池的大小做任何的限制。

```
[obmalloc.c]
#ifdef WITH_MEMORY_LIMITS
#ifndef SMALL_MEMORY_LIMIT
#define SMALL_MEMORY_LIMIT  (64 * 1024 * 1024)  /* 64 MB -- more? */
#endif
#endif

#ifdef WITH_MEMORY_LIMITS
#define MAX_ARENAS      (SMALL_MEMORY_LIMIT / ARENA_SIZE)
#endif
```

尽管我们在前面花费了大量篇幅介绍 arena，同时也看到 arena 是 Python 小块内存池的最上层结构，所有 arena 的集合实际就是小块内存池。然而在实际的使用中，Python 并不直接与 arenas 和 arena 打交道。当 Python 申请内存时，最基本的操作单元并不是 arena，而是 pool。唉，绕来绕去的，我都觉得快晕了，没办法，兄弟，挺住，后面还有很多呢☺。

举个例子，当我们申请一个 28 字节的内存时，Python 内部会在内存池中寻找一块能满足需求的 pool，从中取出一个 block 返回，而不会去寻找 arena。这实际上是由 pool 和 arena 自身的属性决定的。在 Python 中，pool 是一个有 size 概念的内存管理抽象体，一个 pool 中的 block 总是有确定的大小，这个 pool 总是和某个 size class index 对应，还记得 pool_head 中的那个 szidx 么？而 arena 是没有 size 概念的内存管理抽象体，这就意味着，同一个 arena，在某个时刻，其内的 pool 集合可能都是管理的 32 字节的 block；而到了另一时刻，由于系统需要，这个 arena 可能被重新划分，其中的 pool 集合可能改为管理 64 字节的 block 了，甚至 pool 集合中一半管理 32 字节，一半管理 64 字节。这就决定了在进行内存分配和销毁时，所有的动作都是在 pool 上完成的。

内存池中的 pool，不仅是一个有 size 概念的内存管理抽象体，而且，更进一步的，它还是一个有状态的内存管理抽象体。一个 pool 在 Python 运行的任何一个时刻，总是处于

以下三种状态的一种：

- used 状态：pool 中至少有一个 block 已经被使用，并且至少有一个 block 还未被使用。这种状态的 pool 受控于 Python 内部维护的 usedpools 数组；
- full 状态：pool 中所有的 block 都已经被使用，这种状态的 pool 在 arena 中，但不在 arena 的 freepools 链表中；
- empty 状态：pool 中所有的 block 都未被使用，处于这个状态的 pool 的集合通过其 pool_header 中的 nextpool 构成一个链表，这个链表的表头就是 arena_object 中的 freepools；

图 16-6 给出了一个 arena 中包含三种状态的 pool 的集合的一个可能状态。

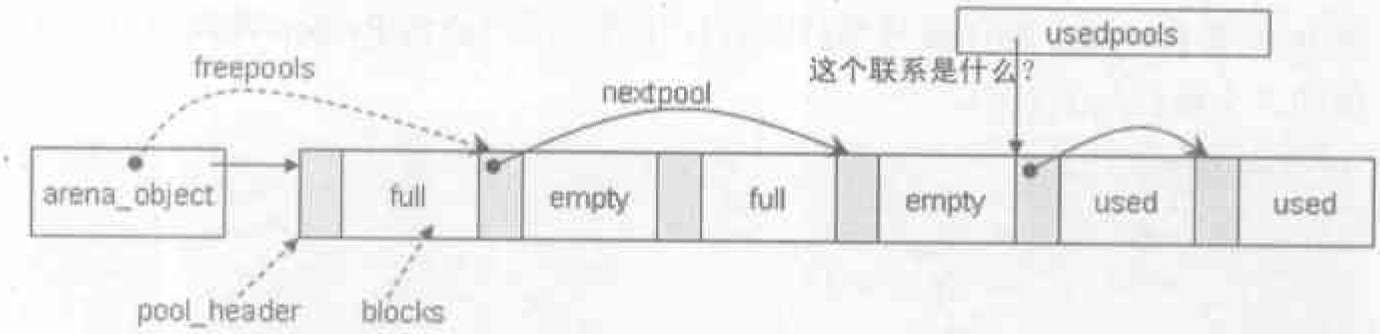



图 16-6 某个时刻 aerna 中 pool 集合的可能状态

请注意，arena 中处于 full 状态的 pool 是各自独立的，并没有像其他的 pool 一样会链接成链表。

在图 16-6 中，我们看到所有处于 used 状态的 pool 都被置于 usedpools 的控制之下。Python 内部维护的 usedpools 数组是一个非常巧妙的实现，维护着所有的处于 used 状态的 pool。当申请内存时，Python 就会通过 usedpools 寻找到一块可用的（处于 used 状态的）pool，从中分配一个 block。从这个简要的叙述中，我们已经可以看到，一定有一个与 usedpools 相关联的机制，完成从申请的内存的大小到 size class index 之间的转换，否则 Python 也就无法寻找到最合适的 pool 了。这种机制与 usedpools 的结构有密切的关系，我们来看一看 usedpools 的结构。

```
[obmalloc.c]
typedef uchar block;

#define PTA(x)  ((poolp )((uchar *)&(usedpools[2*(x)]) - 2*sizeof(block *)))
#define PT(x)   PTA(x), PTA(x)

static poolp usedpools[2 * ((NB_SMALL_SIZE_CLASSES + 7) / 8) * 8] = {
    PT(0), PT(1), PT(2), PT(3), PT(4), PT(5), PT(6), PT(7)
#if NB_SMALL_SIZE_CLASSES > 8
    , PT(8), PT(9), PT(10), PT(11), PT(12), PT(13), PT(14), PT(15)
    ......
#endif
}
```

其中的 NB_SMALL_SIZE_CLASSES 指明了在当前的配置之下，一共有多少个 size class。

```
[obmalloc.c]
#define NB_SMALL_SIZE_CLASSES   (SMALL_REQUEST_THRESHOLD / ALIGNMENT)
```

这个数组的定义有些怪异，别急，待我们用一幅图来展示这个怪异的 usedpools 数组，如图 16-7 所示。

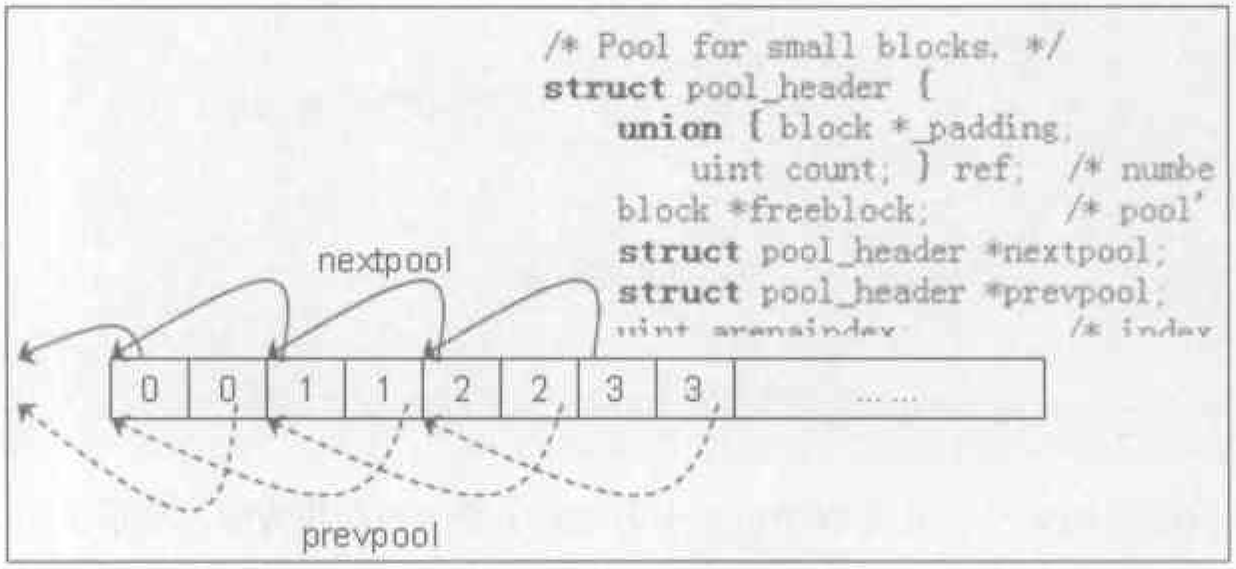



图 16-7 usedpools 数组

这样看上去似乎仍然摸不着头脑，别急，我们来考虑一下当申请 28 个字节时的情形。前面我们提到，Python 会首先获得 size class index，通过 size = (uint )(nbytes - 1) >> ALIGNMENT_SHIFT，得到 size class index 为 3。在 usedpools 中，寻找第 3+3=6 个元素，发现 usedpools[6]的值是指向 usedpools[4]的地址。有些迷惑了，对吧？好了，现在对照 pool_header 的定义来看一看 usedpools[6]->nextpool 这个指针指向哪里了呢？是从 usedpools[6]（即 usedpools+4）开始向后偏移 8 个字节（一个 ref 的大小加上一个 freeblock 的大小）后的内存，不正是 usedpools[6]的地址（即 usedpools+6）吗？这是 Python 内部使用的一个 trick。

想象一下，当我们手中有一个 size class 为 32 字节的 pool，想要将其放入这个 usedpools 中时，需要怎么做呢？从上面的描述可以看到，只需要进行 usedpools[i+i]->nextpool = pool 即可，其中 i 为 size class index，对应于 32 字节，这个 i 为 3。当下次需要访问 size class 为 32 字节（size class index 为 3）的 pool 时，只需要简单地访问 usedpool[3+3]就可以得到了。Python 正是使用这个 usedpools 快速地从众多的 pool 中快速地寻找到一个最适合当前内存需求的 pool，从中分配一块 block。

在我们即将看到的 PyObject_Malloc 代码中，Python 利用了 usedpools 的巧妙结构，通过简单的判断来发现与某个 class size index 对应的 pool 是否在 usedpools 中存在。下面是 PyObject_Malloc 中进行这个判断的代码。

```
[obmalloc.c]
void* PyObject_Malloc(size_t nbytes)
{
    block *bp;
    poolp pool;
    poolp next;
    uint size;
    if ((nbytes - 1) < SMALL_REQUEST_THRESHOLD) {
        LOCK();
        //获得 size class index
        size = (uint )(nbytes - 1) >> ALIGNMENT_SHIFT;
        pool = usedpools[size + size];
        //usedpools 中有可用的 pool
        if (pool != pool->nextpool) {
            ……//usedpools 中有可用的 pool
        }

        …… //usedpools 中无可用 pool，尝试获取 empty 状态 pool
}
```

在图 16-8 中，还是以申请大小为 28 字节的内存块为例，展示了 `pool != pool->nextpool` 为什么能够工作的原因。我们在 Python 的源代码中添加了代码，使得 Python 在第一次申请 size class index 为 3 的内存块时发生中断，以便形象地观察这时 usedpools 的内存布局。图 16-8 左侧粗体显示的地址即是 `pool` 和 `pool->nextpool` 的值。

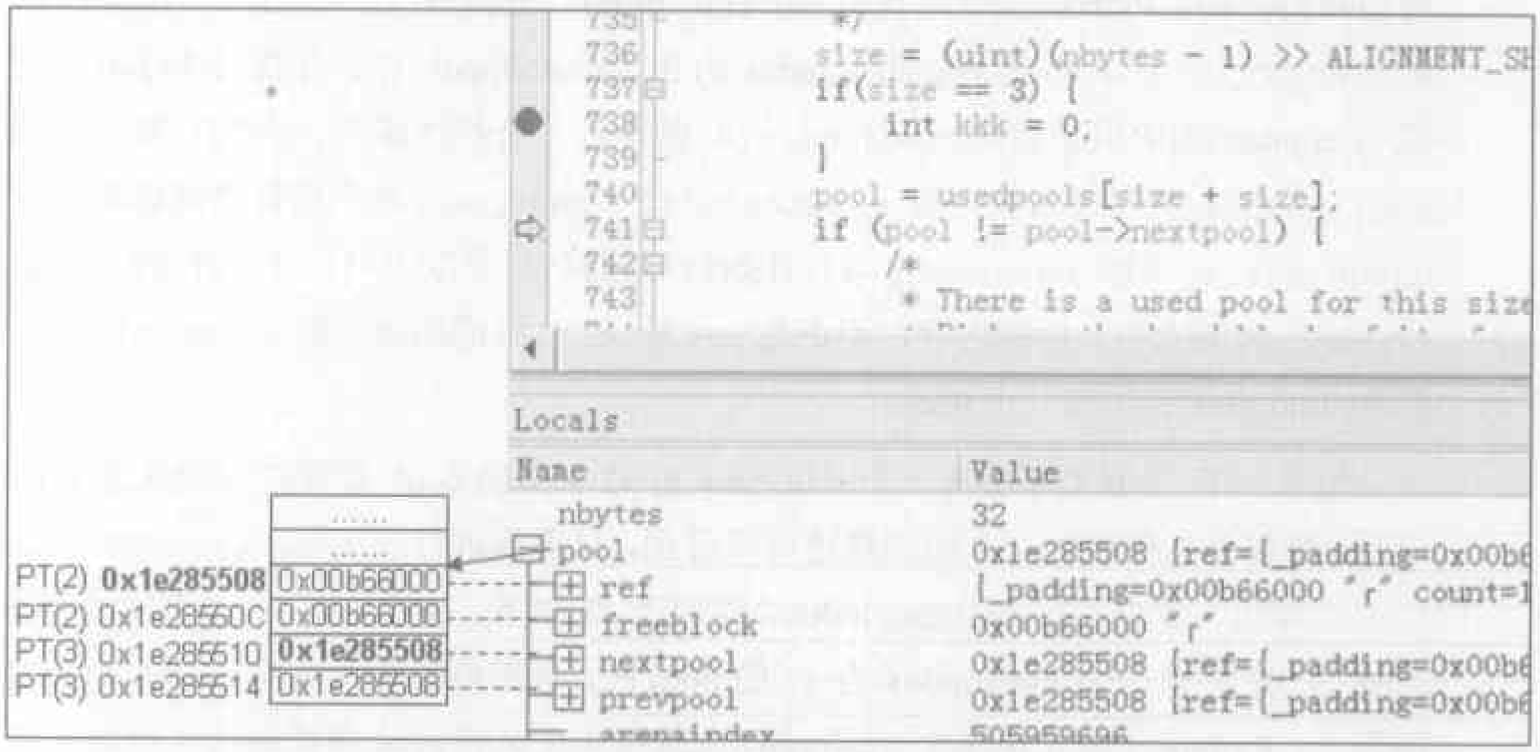


图 16-8 第一次申请 28 字节的内存块时 usedpools 中相关的内存布局

## 16.2.4.2 Pool 的初始化

当 Python 启动之后，在 usedpools 这个小块空间内存池中，并不存在任何可用的内存，准确地说，不存在任何可用的 pool。在这里，Python 采用了延迟分配的策略，即当我们确实开始申请小块内存时，Python 才开始建立这个内存池。正如前面所提到的，当申请 28

个字节的内存时，Python 实际上将申请 32 字节的内存。Python 首先会根据 32 字节对应的 class size index（3）在 usedpools 中对应的位置查找，如果发现在对应的位置后并没有链接任何可用的 pool，Python 会从 `usable_arenas` 链表中的第一个“可用的”arena 中获得一个 pool。需要特别注意的是，当前获得的 arena 中包含的这些 pools 可能并不属于同一个 class size index。

考虑一下这样的情况，当申请 32 字节内存时，从“可用的”arena 中取出其中一个 pool 用作 32 字节的 pool。当下一次内存分配请求分配 64 字节的内存时，Python 可以直接使用当前“可用的”arena 的另一个 pool 即可。这正如我们前面所说，arena 没有 size class 的属性，而 pool 才有（见代码清单 16-4）。

**代码清单 16-4**

```
[obmalloc.c]
void * PyObject_Malloc(size_t nbytes)
{
    block *bp;
    poolp pool;
    poolp next;
    uint size;

  if ((nbytes - 1) < SMALL_REQUEST_THRESHOLD) {
      LOCK();
      size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;
      pool = usedpools[size + size];
      if (pool != pool->nextpool) {
          …… //usedpools 中有可用的 pool
      }
      //usedpools 中无可用 pool，尝试获取 empty 状态 pool
      //[1]：如果 usable_arenas 链表为空，则创建链表
      if (usable_arenas == NULL) {
          //申请新的 arena_object，并放入 usable_arenas 链表
          usable_arenas = new_arena();
          usable_arenas->nextarena = usable_arenas->prevarena = NULL;
      }

      //[2]：从 usable_arenas 链表中第一个 arena 的 freepools 中抽取一个可用的 pool
      pool = usable_arenas->freepools;
      if (pool != NULL) {
          usable_arenas->freepools = pool->nextpool;
          //[3]：调整 usable_arenas 链表中第一个 arena 中的可用 pool 数量
          //如果调整后数量为 0，则将该 arena 从 usable_arenas 链表中摘除
          --usable_arenas->nfreepools;
          if (usable_arenas->nfreepools == 0) {
              usable_arenas = usable_arenas->nextarena;
              if (usable_arenas != NULL) {
                  usable_arenas->prevarena = NULL;
              }
          }
      init_pool:
```

```
    ......
}
```

可以看到，如果开始时 usable_arenas 为空，那么 Python 会在代码清单 16-4 的[1]处通过 new_arena 申请一个 arena，开始构建 usable_arenas 链表，这里就是我们刚才遗留下来的问题的答案，在这里，一个脱离了 unused_arena_objects 并转变为“可用”的 arena 被纳入了 usable_arenas 的控制。在代码清单 16-4 的[2]处，Python 会尝试从 usable_arenas 链表中的第一个 arena 所维护的 pool 集合中取出一个可用的 pool。如果成功地取出了这个 pool，那么在[3]处，Python 会进行一些维护信息的更新工作，甚至在当前 arena 中可用的 pool 已经用完之后，将该 arena 从 usable_arenas 链表中摘除。那位客官说了，你不能摘下来就一了百了啊，摘下来之后这块内存不就失去控制了？别急，别忘了还有个 arenas 数组啊，孙猴子再厉害，也逃不出如来佛的魔爪的☺。

需要注意的是，在代码清单 16-4 的[2]处的判断表明获得 pool 有可能失败，那么在什么情况下，一个 arena 中的 freepools 会是 NULL 呢，呃，回忆一下前面对 new_arena 的剖析，没错，在那里，new_arena 准备返回的 arena 的 freepools 就是为 NULL 的。那么在[2]处发现 pool 为 NULL 时 Python 会怎么处理呢，我们把这个话题放到后面。

### 初始化之一

好了，现在我们手里有了一块用于 32 字节内存分配的 pool，为了以后提高内存分配的效率，我们需要将这个 pool 放入到 usedpools 中。这一步，叫做 init pool（见代码清单 16-5）。

代码清单 16-5

```
[obmalloc.c]
#define ROUNDUP(x)       (((x) + ALIGNMENT_MASK) & ~ALIGNMENT_MASK)
#define POOL_OVERHEAD        ROUNDUP(sizeof(struct pool_header))
void * PyObject_Malloc(size_t nbytes) {
......
init_pool:
            //[1]: 将 pool 放入 usedpools 中
            next = usedpools[size + size]; /* == prev */
            pool->nextpool = next;
            pool->prevpool = next;
            next->nextpool = pool;
            next->prevpool = pool;
            pool->ref.count = 1;
            //[2]: pool 在之前就具有正确的 size 结构，直接返回 pool 中的一个 block
            if (pool->szidx == size) {
                bp = pool->freeblock;
                pool->freeblock = *(block **)bp;
                UNLOCK();
                return (void *)bp;
            }
            //[3]: 初始化 pool header，将 freeblock 指向第二个 block，返回第一个 block
```

```
        pool->szidx = size;
        size = INDEX2SIZE(size);
        bp = (block *)pool + POOL_OVERHEAD;
        pool->nextoffset = POOL_OVERHEAD + (size << 1);
        pool->maxnextoffset = POOL_SIZE - size;
        pool->freeblock = bp + size;
        *(block **)(pool->freeblock) = NULL;
        UNLOCK();
        return (void *)bp;
......
}
```

在代码清单 16-5 的[1]处，Python 将得到的 pool 放入了 usedpools 中。当一个从未被使用的 pool（也就是由 new_arena 返回的 aerna 中的 pool）被链入 usedpools 中时，从后面的分析可以看到，其 szidx 是被设为了 0xFFFF 的，所以这时 init pool 的动作会执行[3]，而不会执行[2]。只有当一个 pool 从 empty 状态重新转为 used 状态之后，由于这时 szidx 还是其转为 empty 状态之前的 szidx，所以才有可能执行[2]。

在什么样的情况下才会发生一个 pool 从 empty 状态转换为 used 状态呢？假设申请的内存的 size class index 为 i，且 usedpools[i+i]处没有处于 used 状态的 pool，同时在 Python 维护的全局变量 freepools 中还有处于 empty 的 pool，那么位于 freepools 所维护的 pool 链表头部的 pool 将被取出来，放入 usedpools 中，并从其内部分配一块 block。同时，这个 pool 也就从 empty 状态转换到了 used 状态。下面我们看一看这个行为在代码中是如何体现的。

```
[obmalloc.c]
    ......
    pool = usable_arenas->freepools;
    if (pool != NULL) {
        usable_arenas->freepools = pool->nextpool;
        ...... //调整 usable_arenas->nfreepools 和 usable_arenas 自身
        [init_pool]
    }
```

其中[init_pool]处引用的是前面剖析的关于 init pool 的代码。需要注意的是，虽然一个 pool 从 empty 状态转为 used 状态时，携带了有效的 szidx 信息，但是这只是上一次 pool 被使用时的信息。只有当当前内存分配动作对应的 size class index 与这个 szidx 完全一致时，才会执行代码清单 16-5 的[2]，否则，Python 还是会照常进行[3]，以重新对 pool 进行初始化。

## 初始化之二

我们现在可以来看看，当 PyObject_Malloc 从 new_arena 中得到一个新的 arena 后，是怎么样来初始化其中的 pool 集合，并最终完成 PyObject_Malloc 函数的分配一个 block 这个终极任务的（见代码清单 16-6）。

**代码清单 16-6**

```
[obmalloc.c]
#define DUMMY_SIZE_IDX      0xffff  /* size class of newly cached pools */
void * PyObject_Malloc(size_t nbytes)
{
    block *bp;
    poolp pool;
    poolp next;
    uint size;
    ……
    //[1]: \\从 arena 中取出一个新的 pool
    pool = (poolp)usable_arenas->pool_address;
    pool->arenaindex = usable_arenas - arenas;
    pool->szidx = DUMMY_SIZE_IDX;
    usable_arenas->pool_address += POOL_SIZE;
    --usable_arenas->nfreepools;

    if (usable_arenas->nfreepools == 0) {
        /* Unlink the arena:  it is completely allocated. */
        usable_arenas = usable_arenas->nextarena;
        if (usable_arenas != NULL) {
            usable_arenas->prevarena = NULL;
        }
    }
    goto init_pool;
    ……
}
```

Python 首先在代码清单 16-6 的[1]处从新申请的 arena 中取出一个崭新的 pool，然后设置 pool 中的 arenaindex，这个 index 实际上就是 pool 所在的 arena 位于 arenas 所指的数组中的序号，这个东西有什么用呢，用处大着呢。它虽然不能医治跌打损伤，却能用来判断一个 block 是否在某个 pool 中。还记得我们在考察 pool 管理 block 集合时看到的那个引起 freeblock 舞动的 PyObject_Free 吗，里面就用到这个 arenaindex。

```
[obmalloc.c]
//P 为指向一个 block 的指针，pool 为指向一个 pool 的指针
int Py_ADDRESS_IN_RANGE(void *P, poolp pool)
{
    return pool->arenaindex < maxarenas &&
        (uptr)P - arenas[pool->arenaindex].address < (uptr)ARENA_SIZE &&
        arenas[pool->arenaindex].address != 0;
}
```

在实际发布的 Python 2.5 中，PyObject_Free 里实际上使用的是宏版本的 Py_ADDRESS_IN_RANGE，而并非这里给出的函数版本 Py_ADDRESS_IN_RANGE，但是它们的代码都是一样的。

随后 Python 将新得到的 pool 的 szidx 设置为 0xffff，以表示这家伙以前从来没管理过 block 集合。接着，Python 还调整了刚获得的 arena 中的 pools 集合，甚至可能调整 usable_arenas。

在做完这些之后，Python 会通过 `goto` 直接跳到 init pool 的地方，完成将 `pool` 放入 usedpools 中的任务。

无论什么开源的项目，内存管理都是最繁琐，最能体现“细节是魔鬼”的地方，在申请内存的过程中，还有一些细节这里就不再一一涉及了，不过最后我们还是要给出 `PyObject_Malloc` 的总体的结构。同时，强烈建议您打开代码阅读的工具，一头扎进 `PyObject_Malloc` 的细节中。

```
[obmalloc.c]
void * PyObject_Malloc(size_t nbytes)
{
    block *bp;
    poolp pool;
    poolp next;
    uint size;

    //如果申请的内存小于 SMALL_REQUEST_THRESHOLD，使用 Python 的小块内存的内存池
    //否则，转向 malloc
    if ((nbytes - 1) < SMALL_REQUEST_THRESHOLD) {
        //根据申请内存的大小获得对应的 size class index
        size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;
        pool = usedpools[size + size];
        //如果 usedpools 中可用的 pool，使用这个 pool 来分配 block
        if (pool != pool->nextpool) {
            ......//在 pool 中分配 block
             //分配结束后，如果 pool 中的 block 都被分配了，将 pool 从 usedpools 中摘除
             next = pool->nextpool;
             pool = pool->prevpool;
             next->prevpool = pool;
             pool->nextpool = next;
             return (void *)bp;
        }

        //usedpools 中没有可用的 pool，从 usable_arenas 中获取 pool
        if (usable_arenas == NULL) {
             //usable_arenas 中没有就"可用"的 arena，开始申请 arena
             usable_arenas = new_arena();
             usable_arenas->nextarena = usable_arenas->prevarena = NULL;
        }

        //从 usable_arenas 的第一个 arena 中获取一个 pool
        pool = usable_arenas->freepools;
        if (pool != NULL) {
init pool:
            //获取 pool 成功，进行 init pool 的动作，将 pool 放入 used_pools 中，
            //并返回分配得到的 block
            ......
        }

    //获取 pool 失败，对 arena 中的 pool 集合进行初始化，
    //然后转入 goto 到 init pool 的动作处，初始化一个特定的 pool
```

```
    ......
        goto init_pool;
    }

redirect:
    //如果申请的内存不小于 SMALL_REQUEST_THRESHOLD，使用 malloc
    if (nbytes == 0)
        nbytes = 1;
    return (void *)malloc(nbytes);
}
```

### 16.2.4.3 block 的释放

考察完了对 block 的分配，是时候来看看对 block 的释放了。对 block 的释放实际上就是将一块 block 归还给 pool，我们已经知道，pool 可能有 3 种状态，在分别处于 3 种状态，它们各自的位置是不同的。

当我们释放一个 block 后，可能会引起 pool 的状态的转变，这种转变可分为两种情况：

- used 状态转变为 empty 状态
- full 状态转变为 used 状态

当然，更多的情况是 pool 中尽管收回了一个 block，但是它仍然处于 used 状态。这是最简单的情况，我们从这个最简单的情况说起。

```
[obmalloc.c]
void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);
    if (Py_ADDRESS_IN_RANGE(p, pool)) {
        //设置离散自由 block 链表
        *(block **)p = lastfree = pool->freeblock;
        pool->freeblock = (block *)p;
        if (lastfree) { //lastfree 有效，表明当前 pool 不是处于 full 状态
            if (--pool->ref.count != 0) { //pool 不需要转换为 empty 状态
                return;
            }
            ......
        }
        ......
    }

    //待释放的内存在 PyObject_Malloc 中是通过 malloc 获得的
    //所以要归还给系统
    free(p);
}
```

在 pool 的状态保持 used 状态这种情况下，Python 仅仅将 block 重新放入到自由 block 链表中，并调整了 pool 中的 ref.count 这个引用计数，确实非常简单。

如果释放 block 之前，block 所属的 pool 处于 full 状态呢？这种情况也比较简单，仅仅是将 pool 重新链回到 usedpools 中即可，看下面的代码：

```
[obmalloc.c]
void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);
    if (Py_ADDRESS_IN_RANGE(p, pool)) {
        ……
        //当前pool处于full状态，在释放一块block后，需将其转换为used状态，并重新
        //链入usedpools的头部
        --pool->ref.count;
        size = pool->szidx;
        next = usedpools[size + size];
        prev = next->prevpool;
        /* insert pool before next:   prev <-> pool <-> next */
        pool->nextpool = next;
        pool->prevpool = prev;
        next->prevpool = pool;
        prev->nextpool = pool;
        return;
    }
    ……
}
```

最复杂的情况发生在 pool 在收回 block 前后状态从 used 状态转为 empty 状态的情形下，我们来看看 Python 的处理。首先 Python 要做的是将 empty 状态的 pool 链入到 freepools 中去（见代码清单 16-7）。

**代码清单 16-7**

```
[obmalloc.c]
void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);
    if (Py_ADDRESS_IN_RANGE(p, pool)) {
        *(block **)p = lastfree = pool->freeblock;
        pool->freeblock = (block *)p;
        if (lastfree) {
            struct arena_object* ao;
```

```
            uint nf; //ao->nfreepools
            if (--pool->ref.count != 0) {
                return;
            }
            //[1]: 将 pool 放入 freepools 维护的链表中
            ao = &arenas[pool->arenaindex];
            pool->nextpool = ao->freepools;
            ao->freepools = pool;
            nf = ++ao->nfreepools;
            ......
        }
        ......
}
......
}
```

代码清单 16-7 的[1]完成了将 empty 状态的 `pool` 放入 `freepools` 维护的链表中的工作，似乎这样就可以了，一切都能够正常运转了，所有的内存始终都在掌握之中。

确实，在 Python 2.5 之前，包括 2.4，Python 就是这么做的，但是这样做隐藏着一个类似于内存泄漏的问题。很多人也都意识到这个问题，但都没有太大的动力去修改，因为这种情况只在极少数情况下会发生。

这个问题就是：Python 的 arena 从来不释放 `pool`。这个问题为什么会引起类似于内存泄漏的现象呢。考虑这样一种情形，申请 10×1024×1024 个 16 字节的小内存，这就意味着必须使用 160MB 的内存，由于 Python 没有默认将前面提到的限制内存池的 `WITH_MEMORY_LIMITS` 编译符号打开，所以 Python 会完全使用 arena 来满足你的需求。这都没有问题，关键的问题在于过了一段时间，你将所有 16 字节的内存都释放了，这些内存都回到 arena 的控制中，似乎也没有问题。但是问题恰恰就在这时出现了。因为 arena 始终不会释放它维护的 `pool` 集合，所以这 160MB 的内存始终被 Python 占用，如果以后程序运行中再也不需要 160MB 如此巨大的内存，这点内存岂不是就浪费了？

当然，这种情形必须在大量持续申请小内存对象时才会出现，平时大家几乎不会碰到这种情况，所以这个问题也就一直留在了 Python 中，但是在 2004 年的时候，一个叫 Evan Jones 的老兄不能忍受下去了。他在对一些巨大的图做某种算法操作时必须持续申请大量小块内存，这导致 Python 占用的内存冲上 1GB 后就再也掉不下来。想一想，确实相当痛苦，这位老兄一番探索，终于发现问题的根源在 arena 这里，于是一鼓作气，搞出了一套解决方案，并在 PyCon 2005 上做了个报告，引起了强烈的反响。但是 Python 的核心开发团队一直没有将这个 patch 并入到 Python 代码中，一直到 2006 年，才由 Tim Peters（这位老兄的名头在 Python 社区也是响当当的，在 Python 的交互环境下键入 import this，看到这位老兄的名号了吧，the zen of python 的创造者，当然，他在 Python 社区的地位可不是靠这几句广告语获得的）将这个 patch 整理，并入到了 Python 代码中。加入了这个 patch 的 Python 就是我们花了这么多精力剖析的 Python 2.5。

在 Python 2.4 中，实际上对 arena 是没有区分“未使用”和“可用”两种状态的，到了

Python2.5 中，arena 可以将自己维护的 pool 集合释放，返回给操作系统，从而必须从“可用”状态转为“未使用”状态，这也是必须要两种状态的原因。在前面那段代码的[1]之后，当 Python 处理完 pool 之后，就要开始处理 arena 了。

对 arena 的处理实际上分为了 4 种情况。

1. 如果 arena 中所有的 pool 都是 empty 的，释放 pool 集合占用的内存。

```
[obmalloc.c]
void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);
    struct arena_object* ao;
    uint nf;  //ao->nfreepools
    ……
    //将pool放入freepools维护的链表中
    ao = &arenas[pool->arenaindex];
    pool->nextpool = ao->freepools;
    ao->freepools = pool;
    nf = ++ao->nfreepools;
    if (nf == ao->ntotalpools) {
        //调整usable_arenas链表
        if (ao->prevarena == NULL) {
            usable_arenas = ao->nextarena;
        }
        else {
            ao->prevarena->nextarena = ao->nextarena;
        }

        if (ao->nextarena != NULL) {
            ao->nextarena->prevarena = ao->prevarena;
        }
        //调整unused_arena_objects链表
        ao->nextarena = unused_arena_objects;
        unused_arena_objects = ao;
        //释放内存
        free((void *)ao->address);
        //设置address，将arena的状态转为“未使用”
        ao->address = 0;
        --narenas_currently_allocated;
    }
    ……
}
```

可以看见，除了将 arena 维护的 pools 的内存归还给系统之外，Python 还调整了 `usable_arenas` 和 `unused_arena_object` 链表，将 arena 的状态转到了“未使用”状态，以及一些其他的维护工作。

2. 如果之前 arena 中没有了 empty 的 pool，那么在 `usable_arenas` 链表中就找不到该

arena，由于现在 arena 中有了一个 pool，所以需要将这个 aerna 链入到 `usable_arenas` 链表的表头。

3. 若 arena 中的 empty 的 pool 个数为 n，则从 `usable_arenas` 开始寻找 arena 可以插入的位置，将 arena 插入到 `usable_arenas`。这个操作的原因是由于 `usable_arenas` 实际上是一个有序的链表，从表头开始往后，每一个 arena 中的 empty 的 pool 的个数，即 nfreepools，都不能大于前面的 arena，也不能小于前面的 arena。保持这种有序性的原因是分配 block 时，是从 `usable_arenas` 的表头开始寻找可用的 arena 的，这样，就能保证如果一个 arena 的 empty pool 数量越多，它被使用的机会就越少。因此，它最终释放其维护的 pool 集合的内存的机会就越大，这样就能保证多余的内存会被归还给系统。
4. 其他情况，不进行任何对 arena 的处理。

后面三种情况的代码这里就不一一列出了，建议读者自行到 Python 源码中去探索一番。

### 16.2.4.4 内存池全景

前面我们已经提到了，对于一个用 C 开发的庞大的软件，其中的内存管理可能是最复杂最繁琐的部分了，这里我们看到了对不同尺度内存的不同的抽象，看到了这些抽象在各种情况下组成的各式各样的链表，非常复杂。但是，我们还是有可能从一个整体的尺度上把握整个内存池。这就是下面的图 16-9 希望完成的目标。尽管各种不同的链表变幻无常，我们只需记住，所有的内存都在 arenas 的掌控之中。

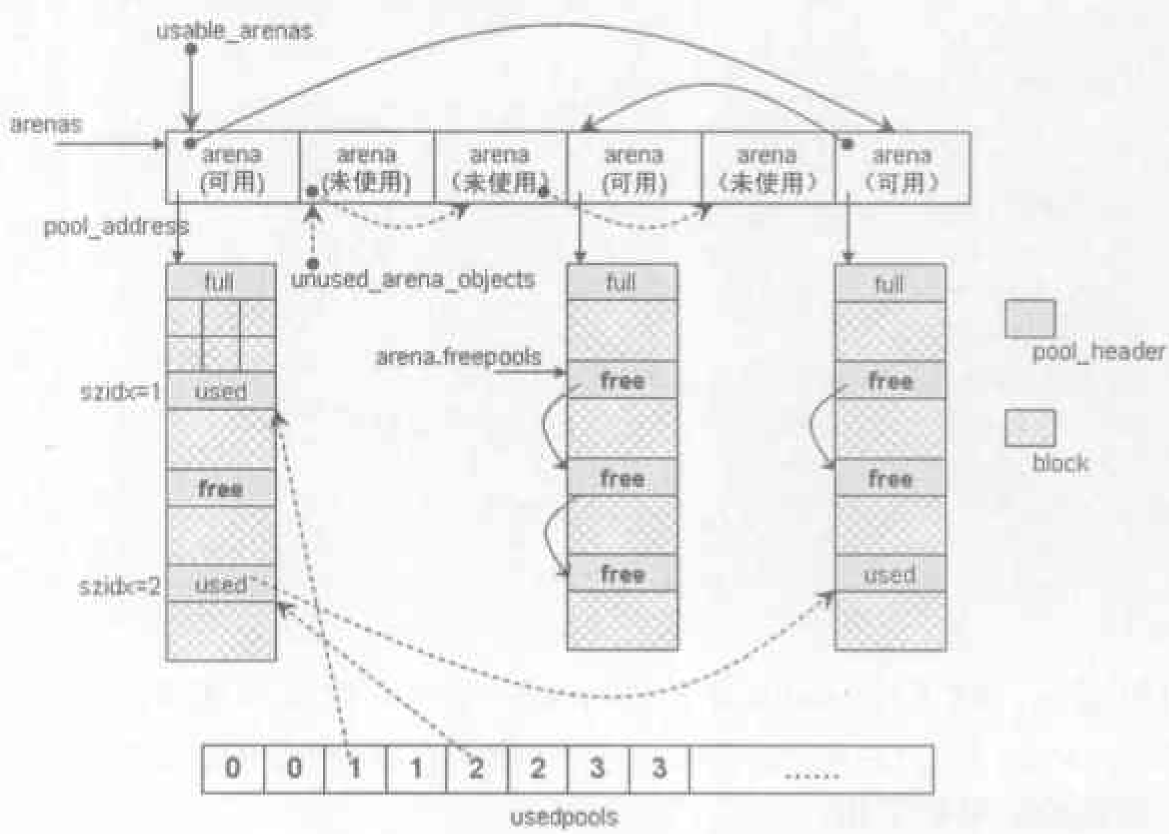



图 16-9 Python 的小块内存的内存池全景

# 16.3 循环引用的垃圾收集

## 16.3.1 引用计数与垃圾收集

尽管学术界对于垃圾收集技术的研究早在上个世纪 60 年代左右就拉开了帷幕，然而受限于当时的软硬件环境，垃圾收集还仅仅是一种看上去很美的技术。挽起裤管，亲自下田，管理一个又一个的字节，不仅是某些应用所必需的，更成为了程序员自身技术水平的象征。然而与手动管理内存相伴随的 bug 也因此绵延几十年，无法根绝。直到上世纪 90 年代初，垃圾收集机制才随着 Java 的兴起，开始逐渐为工业界所接受。时至今日，随着软硬件环境的发展，垃圾收集几乎已经成了现代主流开发语言不可或缺的特性，Java、C#，甚至以替代 C++为目标的 D 语言，都在语言层面引入了垃圾收集机制。

Python 同样也在语言层实现了内存的动态管理，从而将开发人员从管理内存的噩梦中解放出来。然而 Python 中的动态内存管理与 Java、C#有着很大的不同。在 Python 中，大多数对象的生命周期都是通过对象的引用计数来管理的，这一点，在本书前面的分析中，我们已经多次提到。从广义上来说，引用计数也是一种垃圾收集机制，而且也是一种最直观，最简单的垃圾收集技术。虽然引用计数必须在每次分配和释放内存的时候加入管理引用计数的动作，然而与其他主流的垃圾收集技术相比，引用计数方法有一个最大的优点，即实时性，任何内存，一旦没有指向它的引用，就会立即被回收。而其他的垃圾收集技术必须在某种特殊条件下（比如内存分配失败）才能进行无效内存的回收。

引用计数机制所带来的维护引用计数的额外操作与 Python 运行中所进行的内存分配和释放，引用赋值的次数是成正比的，这一点，相对于主流的垃圾回收技术，比如标记——清除（Mark——Sweep）、停止——复制（Stop——Copy）等方法相比，是一个弱点，因为这些技术所带来的额外操作基本上只与待回收的内存数量有关。为了与引用计数机制搭配，在内存的分配和释放上获得最高的效率，Python 因此设计了大量的内存池机制，在第 2 节中我们就看到了小块内存的内存池。而在之前对 Python 对象机制的剖析中，我们看到了对于 `PyIntObject`、`PyStringObject`、`PyDictObject`、`PyListObject` 等等都有与各种对象相关的内存池机制。这些大量使用的面向特定对象的对象内存池机制正是为了竭力弥补引用计数机制的软肋。

如果说执行效率还仅仅是引用计数机制的一个软肋的话，那么很不幸，引用计数还存在着一个致命的弱点，这一点虽然看似很小，然而其存在却几乎宣判了引用计数机制在垃圾收集技术中的“死刑”。也正是由于这一致命的弱点，使得狭义的垃圾收集研究从来没有将引用计数包含在内。这个致命的弱点就是循环引用。

我们知道，引用计数机制非常简单，当一个对象的引用被创建或复制时，对象的引用计数加 1；当一个对象的引用被销毁时，对象的引用计数减 1。如果对象的引用计数减少为 0，那么意味着对象已经不会被任何人使用，可以将其所占用的内存释放。问题的关键就在于，循环引用可以使一组对象的引用计数都不为 0，然而这些对象实际上并没有被任何外部变量引用，它们之间只是互相引用。这意味着不会再有人使用这组对象，应该回收这些对象所占用的内存，然而由于互相引用的存在，每一个对象的引用计数都不为 0，因此这些对象所占用的内存永远不会被回收。图 16-10 展示了 Python 中的一个循环引用。

```
>>> l1 = []
>>> l2 = []
>>> l1.append(l2)
>>> l2.append(l1)
>>> l1
[[[...]]]
>>> l2
[[[...]]]
...
```

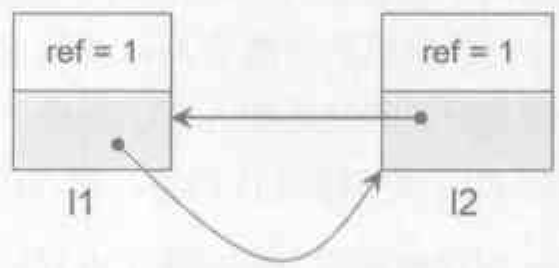



图 16-10 Python 中的循环引用

毫无疑问，这一点是致命的，这与手动进行内存管理所产生的内存泄露毫无区别。虽然在实际中，可以通过某种方法在语言一级保证不出现循环引用，然而这就要求开发者在将精力放到问题域的建模、实现上时，还需要花费额外的精力来精心设计代码结构，以保证不出现循环引用。这一点将立刻把所有的 Python 开发者都推到 Java、C#的阵营中。

要解决这个问题，必须引入其他的垃圾收集技术来打破循环引用，Python 中引入了主流垃圾收集技术中的标记——清除和分代收集两种技术来填补其内存管理机制中最后的也是最致命的漏洞。

## 16.3.2 三色标记模型

无论何种垃圾收集机制，一般都分为两个阶段：垃圾检测和垃圾回收。垃圾检测是从所有的已分配的内存中区别出可以回收的内存和不可回收的内存，而垃圾回收则是使系统重新掌握在垃圾检测阶段所标识出来的可回收内存块。在本节，我们将来看一看标记——清除（Mark——Sweep）方法是如何实现的，并为这个过程建立一个三色标记模型。Python 中的垃圾收集正是基于这个模型完成的。

从具体的实现上来讲，标记——清除方法同样遵循垃圾收集的两个阶段，其简要工作过程如下：

- 寻找根对象（root object）的集合，所谓的 root object 即是一些全局引用和函数栈中的引用。这些引用所用的对象是不可被删除的。而这个 root object 集合也是垃圾检测动作的起点
- 从 root object 集合出发，沿着 root object 集合中的每一个引用，如果能到达某个对象 A，则 A 称为可达的（reachable），可达的对象也不可被删除。这个阶段就是垃圾检测阶段
- 当垃圾检测阶段结束后，所有的对象分为了可达的和不可达的（unreachable）两部分，所有的可达对象都必须予以保留，而所有的不可达对象所占用的内存将被回收，这就是垃圾回收阶段

在垃圾收集动作被激活之前，系统中所分配的所有对象和对象之间的引用组成了一张有向图，其中对象是图中的节点，而对象间的引用是图的边。我们在这个有向图的基础上建立一个三色标注模型，更形象地展示垃圾收集的整个动作。当垃圾收集开始时，我们假设系统中的所有对象都是不可达的，对应在有向图上，即所有的节点都标注为白色。随后，从垃圾收集的动作开始，沿着始于 root object 集合中的某个 object 的引用链，在某个时刻到达了对象 A，那么我们将 A 标记为灰色，灰色表示一个对象是可达的，但是其所包含的引用还没有检查。当我们检查了对象 A 中所包含的所有引用之后，A 将被标记为黑色，以示其包含的所有引用已经被检查过了。显然，这时，A 中引用所引用的对象则被标记为了灰色。假如我们从 root object 集合出发，采用先广搜索的策略，可以想象，灰色节点对象集合就如同一个波的阵面一样，不断向外扩散，随着所有的灰色节点都变为了黑色节点，也就意味着垃圾检测阶段结束了。图 16-11 展示了垃圾收集的某个时刻有向图的一个局部。

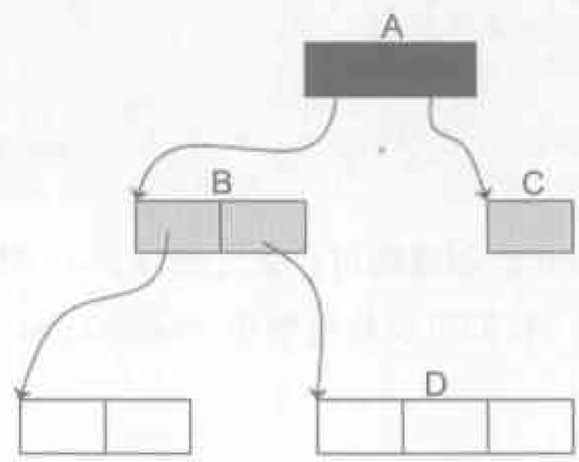



**图 16-11 垃圾收集过程中某个时刻的多个对象组成的有向图**

## 16.4 Python 中的垃圾收集

如前所述，在 Python 中，主要的内存管理手段是引用计数机制，而标记——清除和分代收集只是为了打破循环引用而引入的补充技术。这一事实意味着 Python 中的垃圾收

集只关注可能会产生循环引用的对象。很显然，像 PyIntObject、PyStringObject 这些对象是绝不可能产生循环引用的，因为它们内部不可能持有对其他对象的引用。Python 中的循环引用总是发生在 container 对象之间，所谓 container 对象即是内部可持有对其他对象的引用的对象，比如 list、dict、class、instance 等等。当 Python 的垃圾收集机制运行时，只需要去检查这些 container 对象，而对 PyIntObject、PyStringObject 这些对象则不需理会，这使得垃圾收集带来的开销只依赖于 container 对象的数量，而非所有对象的数量。为了达到这一点，Python 必须跟踪所创建的每一个 container 对象，并将这些对象组织到一个集合中，只有如此，才能将垃圾收集的动作限制在这些对象上。那么 Python 采用了什么结构将这些 container 对象组织在一起呢？Python 采用了一个双向链表，所有的 container 对象在创建之后，都会被插入到这个链表中。

## 16.4.1 可收集对象链表

在对 Python 对象机制的分析中我们已经看到，任何一个 Python 对象都分为两部分，一部分是 PyObject_HEAD，另一部分是对象自身的数据。然而，对于一个需要被垃圾收集机制跟踪的 container 对象而言，这还不够，因为这个对象还必须链入到 Python 内部的可收集对象链表中去。一个 container 对象想要成为一个可收集的对象，则必须加入另外的信息，这个信息位于 PyObject_HEAD 之前，称为 PyGC_Head。

```
[objimpl.h]
typedef union _gc_head {
    struct {
        union _gc_head *gc_next;
        union _gc_head *gc_prev;
        int gc_refs;
    } gc;
    long double dummy;  /* force worst-case alignment */
} PyGC_Head;
```

所以，对于 Python 所创建的可收集 container 对象，其内存分布与我们之前所了解的内存布局是不同的，我们可以从可收集 container 对象的创建过程中窥见其内存分布（见代码清单 16-8）。

**代码清单 16-8**

```
[gcmodule.c]
PyObject* _PyObject_GC_New(PyTypeObject *tp)
{
    PyObject *op = _PyObject_GC_Malloc(_PyObject_SIZE(tp));
    if (op != NULL)
        op = PyObject_INIT(op, tp);
    return op;
}
```

```
#define _PyGC_REFS_UNTRACKED          (-2)
#define GC_UNTRACKED              _PyGC_REFS_UNTRACKED

PyObject* _PyObject_GC_Malloc(size_t basicsize)
{
    PyObject *op;
    //[1]: 为对象本身及 PyGC_Head 申请内存
    PyGC_Head *g = PyObject_MALLOC(sizeof(PyGC_Head) + basicsize);
    g->gc.gc_refs = GC_UNTRACKED;//[2]
    generations[0].count++; /* number of allocated GC objects */
    //[3]
    if (generations[0].count > generations[0].threshold &&
        enabled &&
        generations[0].threshold &&
        !collecting &&
        !PyErr_Occurred()) {
        collecting = 1;
        collect_generations();
        collecting = 0;
    }
    op = FROM_GC(g);//[4]
    return op;
}
```

从代码清单 16-8 的[1]处可以清楚地看到，当 Python 为可收集的 container 对象申请内存空间时，为 PyGC_Head 也申请了内存空间，并且其位置在 container 对象之前。所以我们现在对于 PyListObject、PyDictObject 等对象的内存分布的推测应该变成如图 16-12 所示的情形。需要注意，在申请内存时，使用的是 PyObject_MALLOC，这将最终调用我们在上一节花费巨大精力分析的 PyObject_Malloc。

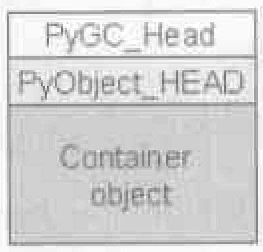



图 16-12　被垃圾收集机制监控的 container 对象

在可收集 container 对象的内存分布中，内存分为三个部分，首先的一块用于垃圾收集机制，然后紧跟着的是 Python 中所有对象都会有的 PyObject_HEAD，最后才是属于 container 对象自身的数据。这里的 Container Object 既可以是 PyDictObject，也可以是 PyListObject，等等。

在 PyGC_Head 部分，除了两个用于建立链表结构的前向和后向指针外，还有一个 gc_ref，在代码清单 16-8 的[2]处我们看到，这个值被初始化为 GC_UNTRACED。这个变量对于垃圾收集的运行至关重要，但是现在，在深入这个变量以及垃圾收集之前，我们还需要了解其他一些事实，这里我们先将其放下。同样对于[3]，我们这里也先不剖析，不过别急，马上我们就会回来。

当垃圾收集机制运行期间，我们需要在一个可收集的 container 对象的 PyGC_Head 部分和 PyObject_HEAD 部分来回转换。更清楚地说，某些时候，我们持有一个对象 A 的 PyObject_HEAD 的地址，但是我们需要根据此地址获得 A 的 PyGC_Head 的地址；而在某些时候，我们又需要进行这一动作的逆运算。Python 提供了在两个地址之间的转换算法，代码清单 16-8 的[4]处使用的是从 PyGC_Head 地址转换为 PyObject_HEAD 地址的算法。

```
[gcmodule.c]
/* Get an object's GC head */
#define AS_GC(o) ((PyGC_Head *)(o)-1)
/* Get the object given the GC head */
#define FROM_GC(g) ((PyObject *)(((PyGC_Head *)g)+1))

[objimpl.h]
#define _Py_AS_GC(o) ((PyGC_Head *)(o)-1)
```

在 PyGC_Head 中，出现了用于建立链表的两个指针，只有将创建的可收集 container 对象链接到 Python 内部维护的可收集对象链表中，Python 的垃圾收集机制才能跟踪和处理这个 container 对象。但是我们发现，在创建可收集 container 对象之时，并没有立即将这个对象链接到链表中。实际上，这个动作是发生在创建某个 container 对象的最后一步，从 PyDictObject 的创建过程，我们可以清楚地看到这一点。

```
[dictobject.c]
PyObject* PyDict_New(void)
{
    register dictobject *mp;
    ……
    mp = PyObject_GC_New(dictobject, &PyDict_Type);
    ……
    _PyObject_GC_TRACK(mp);
    return (PyObject *)mp;
}
```

在创建 container 对象的最后一步，Python 通过_PyObject_GC_TRACD 将所创建的 container 对象链接到了 Python 中的可收集对象链表中。

```
[objimpl.h]
#define _PyObject_GC_TRACK(o) do { \
    PyGC_Head *g = _Py_AS_GC(o); \
    if (g->gc.gc_refs != _PyGC_REFS_UNTRACKED) \
        Py_FatalError("GC object already tracked"); \
    g->gc.gc_refs = _PyGC_REFS_REACHABLE; \
    g->gc.gc_next = _PyGC_generation0; \
    g->gc.gc_prev = _PyGC_generation0->gc.gc_prev; \
    g->gc.gc_prev->gc.gc_next = g; \
    _PyGC_generation0->gc.gc_prev = g; \
    } while (0);
```

前面我们说过，Python 会将自己的垃圾收集机制限制在其维护的可收集对象链表上，因为所有的循环引用一定是发生在这个链表中的一群对象之间。在_PyObject_GC_TRACK 之后，我们所创建的 container 对象也就置于 Python 垃圾收集机制的掌控之中了。

同样，Python 还提供了将一个 container 对象从链表中摘除的方法，显然，这个方法应该在对象被销毁时调用。

```
[objimpl.h]
#define _PyObject_GC_UNTRACK(o) do { \
    PyGC_Head *g = _Py_AS_GC(o); \
    assert(g->gc.gc_refs != _PyGC_REFS_UNTRACKED); \
    g->gc.gc_refs = _PyGC_REFS_UNTRACKED; \
    g->gc.gc_prev->gc.gc_next = g->gc.gc_next; \
    g->gc.gc_next->gc.gc_prev = g->gc.gc_prev; \
    g->gc.gc_next = NULL; \
    } while (0);
```

很明显，_PyObject_GC_UNTRACK 仅仅是_PyObject_GC_TRACK 的逆运算而已。在图 16-13 中，我们展示了 Python 运行过程的某个时刻，所建立起来的可收集对象链表。

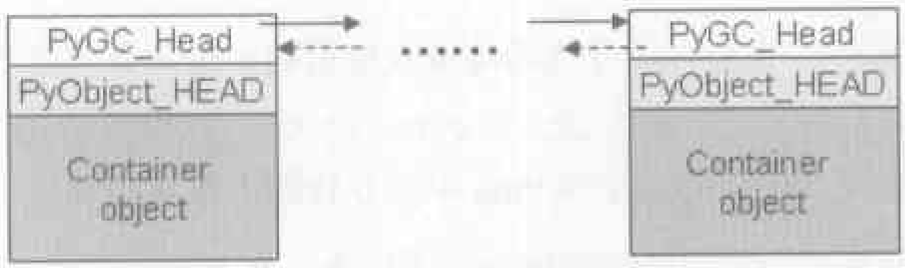



图 16-13 可收集对象链表

## 16.4.2 分代的垃圾收集

分代的垃圾收集技术是在上个世纪 80 年代初发展起来的一种垃圾收集机制，一系列的研究表明，无论使用何种语言开发，无论开发的是何种类型、何种规模的程序，都存在这样一点相同之处。即，一定比例的内存块的生存周期都比较短，通常是几百万条机器指令的时间，而剩下的内存块，其生存周期会比较长，甚至会从程序开始一直持续到程序结束。研究表明，对于不同的语言，不同的应用程序，这个比例会在 80%到 98%之间游走。

这一发现对于垃圾收集技术有着重要的意义。从前面的分析我们已经看到，像标记——清除这样的垃圾收集所带来的额外操作实际上与系统中总的内存块的数量是相关的，当需回收的内存块越多时，垃圾检测带来的额外操作就越多，而垃圾回收带来的额外操作就越少；反之，当需回收的内存块越少时，垃圾检测就将比垃圾回收带来更少的额外操作。无论如何，我们可以看到，当系统中使用的内存越少时，整个垃圾收集所带来的额外操作也就越少。为了使垃圾收集的效率提高，基于研究人员所发现的统计规律，我们就可以采用一种以空间换时间的策略。这种以空间换时间的分代收集的技术正是当前支撑着 Java 的关键技术。

这种以空间换时间的总体思想是：将系统中的所有内存块根据其存活时间划分为不同的集合，每一个集合就称为一个“代”，垃圾收集的频率随着“代”的存活时间的增大而

减小，也就是说，活得越长的对象，就越可能不是垃圾，就应该越少去收集。那么这个存活时间是如何来衡量的呢，通常是利用经过了几次垃圾收集动作来衡量，如果一个对象经过的垃圾收集次数越多，那么显然，其存活时间就越长。

举个具体的例子来说，当某些内存块 M 经过了 3 次垃圾收集的洗礼还依然存活时，我们就将 M 划到一个集合 A 中去，而新分配的内存都划到集合 B 中去。当垃圾收集开始工作时，大多数情况下都只对集合 B 进行垃圾回收，而对 A 的回收要等到过了相当长一段时间才进行，这就使得垃圾收集需要处理的内存变少了，效率则得到提高。可以想见，B 中的内存在经过几次收集之后，有一些内存块会被转移到 A 中，而在 A 中，实际上确实会存在一些垃圾，这些垃圾的回收因为这种分代的机制会被延迟。这就是我们所说的以空间换时间的策略。

在 Python 中，也引入了分代的垃圾收集机制，总共有三个“代”。在_PyObject_GC_TRACK 中我们看到了一个名为_PyGC_generation0 的神秘变量，这个变量是 Python 内部维护的一个指针，指向的正是 Python 中第 0 代的内存块集合。

“代”似乎是一个很抽象的概念，实际上，在 Python 中，一个“代”就是一个链表，所有属于同一“代”的内存块都链接在同一个链表中。既然 Python 中总共有 3“代”，那么很显然，Python 中实际是维护了三条链表。更明确地说，一个“代”就是我们在 16.3 节中所提到的一条可收集对象链表，在前面所介绍的链表的基础上，为了支持分代机制，需要的仅仅是一个额外的表头而已。

```
[gcmodule.c]
struct gc_generation {
    PyGC_Head head;
    int threshold; /* collection threshold */
    int count; /* count of allocations or collections of younger
                  generations */
};
```

Python 中有一个维护了三个 gc_generation 结构的数组，通过这个数组控制了三条可收集对象链表，这就是 Python 中用于分代垃圾收集的三个“代”。

```
[gcmodule.c]
#define NUM_GENERATIONS 3
#define GEN_HEAD(n) (&generations[n].head)

/* linked lists of container objects */
static struct gc_generation generations[NUM_GENERATIONS] = {
    /* PyGC_Head,                      threshold,  count */
    {{{GEN_HEAD(0), GEN_HEAD(0), 0}},   700,        0},
    {{{GEN_HEAD(1), GEN_HEAD(1), 0}},   10,         0},
    {{{GEN_HEAD(2), GEN_HEAD(2), 0}},   10,         0},
};

PyGC_Head *_PyGC_generation0 = GEN_HEAD(0);
```

我们在_PyObject_GC_TRACK 中所见的_PyGC_generation0 不偏不斜，指向的正是第 0 代内存集合。图 16-14 展示了用于控制 3 个“代”的 generations。

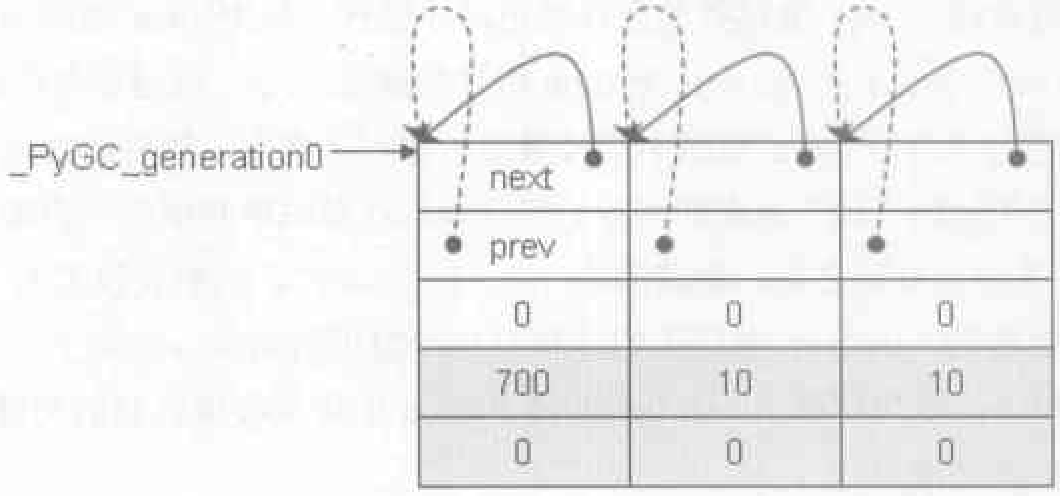



图 16-14 Python 中维护 3 代内存的控制结构

对于每一个 gc_generation，其中的 count 记录了当前这条可收集对象链表中一共有多少个可收集对象。在_PyObject_GC_Malloc 中，我们可以看到，在分配了内存之后，都会进行 generations[0].count++的动作，将第 0 代内存链表中所维护的内存块的数量加 1，这预示着所有新创建的对象实际上都会被加入到第 0 代可收集对象链表中，这一点在_PyObject_GC_TRACK 中被证实了。细心的朋友可能已经发现，这个递增 count 的动作实际上是被提前了，因为直到_PyObject_GC_TRACK 时，所创建的可收集 container 对象才会真正被链接到第 0 代内存链表中。

在 gc_generation 中，threshold 记录了该条可收集对象链表中最多可容纳多少个可收集对象，从 Python 的实现代码中，可以发现，第 0 代链表中最多可以容纳 700 个 container 对象，一旦第 0 代内存链表的 count 超过了 700 这个极限值，则会立刻触发垃圾回收机制。这一点正是_PyObject_GC_Malloc 在代码清单 16-8 的[3]处所表现出来的行为。

```
[gcmodule.c]
static Py_ssize_t collect_generations(void)
{
    int i;
    Py_ssize_t n = 0;

    /* Find the oldest generation (higest numbered) where the count
     * exceeds the threshold.  Objects in the that generation and
     * generations younger than it will be collected. */
    for (i = NUM_GENERATIONS-1; i >= 0; i--) {
        if (generations[i].count > generations[i].threshold) {
            n = collect(i);
            break;
        }
    }
    return n;
}
```

在_PyObject_GC_Malloc 中，虽然是由第 0 代内存链表的越界触发了垃圾收集，但是 Python 会借此时机，对所有“代”内存链表都进行垃圾收集，当然，这只能在与某“代”对应的链表的 count 值越界的条件满足时才进行。从 Python 源码中的注释里我们看到，在 collect_generations 中，Python 将寻找满足 count 值越界条件的最“老”的那一“代”（也就是 generations 数组中序号最高的那一“代”），然后回收这“代”对应的内存和所有比它年轻的“代”对应的内存。但是我们在源码中却明明白白地看见，找到最老的那“代”，并进行处理之后，就潇洒地一个 break 动作，拍拍屁股走人了，比它年轻的“代”根本没处理啊，Python 源码里的注释不是睁眼说瞎话么。实际上，问题的关键出在那个 collect 和它接受的参数上。这个函数是 Python 中垃圾收集机制的关键实现所在，下一节将详细剖析这个函数。

### 16.4.3 Python 中的标记——清除方法

前面我们提到，Python 采用了三代的分代收集机制，如果当前收集的是第 1 代，那么在开始垃圾收集之前，Python 会将比其“年轻”的所有代的内存链表（当然，在这里只有第 0 代）整个地链接到第 1 代内存链表之后，这个操作是通过 gc_list_merge 实现的。

```
[gcmodule.c]
static void gc_list_init(PyGC_Head *list)
{
    list->gc.gc_prev = list;
    list->gc.gc_next = list;
}

static void gc_list_merge(PyGC_Head *from, PyGC_Head *to)
{
    PyGC_Head *tail;
    if (!gc_list_is_empty(from)) {
        tail = to->gc.gc_prev;
        tail->gc.gc_next = from->gc.gc_next;
        tail->gc.gc_next->gc.gc_prev = tail;
        to->gc.gc_prev = from->gc.gc_prev;
        to->gc.gc_prev->gc.gc_next = to;
    }
    gc_list_init(from);
}
```

在我们的例子中，from 就是第 0 代内存链表，而 to 就是第 1 代内存链表。图 16-15 展示了 merge 的结果。

此后的标记——清除算法就将在 merge 之后所得到的那一条内存链表上进行。同时图 16-15 也能说明在 4.2 节末尾的 collect_generations 函数中，为什么 Python 拍拍屁股走人后，还敢大言不惭地说它对符合垃圾回收条件的最“老”的“代”以及所有比它

年轻的“代”都进行了回收。

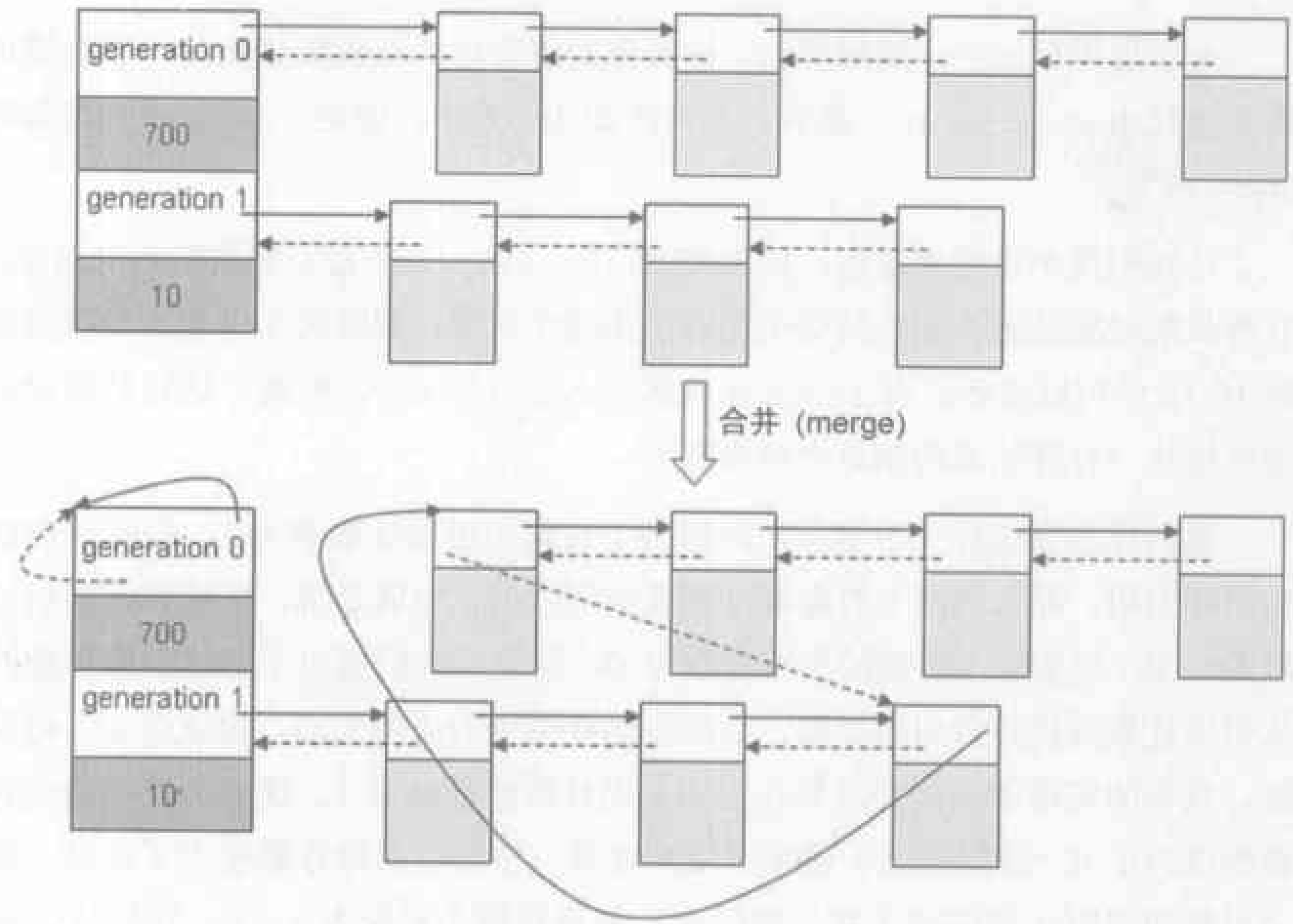



图 16-15 对可收集对象链表的合并操作

在详细剖析 Python 中用于打破循环引用的标记——清除垃圾收集方法之前，需要先建立一个循环引用的最简单的例子，基于这个例子，我们将描述 Python 中使用的标记——清除算法。这个例子与图 16-10 所示的例子相类似，但是不同的是，它多了一个外部引用，如图 16-16 所示。

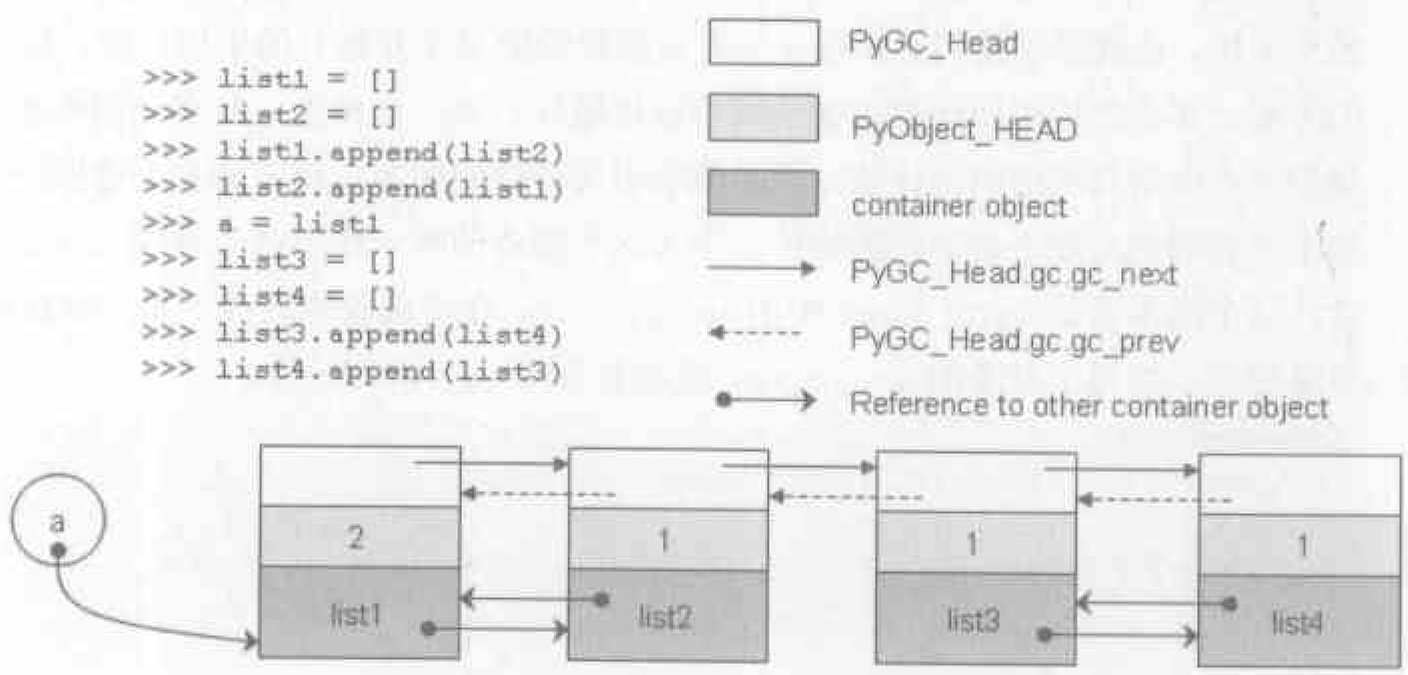



图 16-16 用于演示标记—清除算法的例子

其中，在 `PyObject_HEAD` 部分标出的数值表示对象的引用计数 `ob_refcnt` 的值。

### 16.4.3.1 寻找 Root Object 集合

为了使用标记——清除算法，按照我们前面对垃圾收集算法的一般性描述，首先我们需要寻找出 `root object` 集合。那么在图 16-16 中，哪些 `container` 对象应该属于 root object 呢？

让我们换个角度来思考，前面提到，`root object` 是不能被删除的对象。也就是说，有可收集对象链表外部的某个引用在引用这个对象，删除这个对象会导致错误的行为，从图 16-16 中可以看到只有 `list1` 应该属于 `root object`。但是，这只是观察的结果，应该如何设计一种算法来得到这个结果呢？

我们注意到这样一个事实，如果两个对象的引用计数都为 1，但是仅仅存在它们之间的循环引用，那么这两个对象都是需要被回收的，也就是说，虽然它们的引用计数虽然表现为非 0，但实际上有效的引用计数为 0。这里，我们提出了**有效引用计数**的概念，为了从引用计数获得有效引用计数，必须将循环引用的影响去除，也就是说，将环从引用中摘除，具体的实现就是两个对象各自的引用计数值都减去 1。这样一来，两个对象的引用计数都成为了 0，我们挥去了循环引用的迷雾，使有效引用计数现出了真身。那么如何使两个对象的引用计数都减 1 呢，很简单，假设这两个对象为 `A`、`B`，我们从 `A` 出发，因为它有一个对 `B` 的引用，则将 `B` 的引用计数减 1；然后顺着引用达到 `B`，因为 `B` 有一个对 `A` 的引用，同样将 `A` 的引用减 1，这样，就完成了循环引用对象间环的摘除。

但是这样就引出了一个问题，假设可收集对象链表中的 container 对象 `A` 有一个对对象 `C` 的引用，而 `C` 并不在这个链表中，如果将 `C` 的引用计数减 1，而最后 `A` 并没有被回收，那么显然，`C` 的引用计数被错误地减少了 1，这将导致在未来的某个时刻出现一个对 `C` 的悬空引用。这就要求我们必须在 `A` 没有被删除的情况下复原 `C` 的引用计数，如果采用这样的方案，那么维护引用计数的复杂度将成倍增长。换一个角度，其实我们有更好的做法，我们并不改动真实的引用计数，而是改动引用计数的副本。对于副本无论做任何的改动，都不会影响到对象生命周期的维护，因为这个副本的唯一作用就是寻找 `root object` 集合。这个副本就是 `PyGC_Head` 中的 `gc.gc_ref`。在垃圾收集的第一步，就是遍历可收集对象链表，将每个对象的 `gc.gc_ref` 值设置为其 `ob_refcnt` 值。

```
[gcmodule.c]
static void update_refs(PyGC_Head *containers)
{
    PyGC_Head *gc = containers->gc.gc_next;
    for (; gc != containers; gc = gc->gc.gc_next) {
        assert(gc->gc.gc_refs == GC_REACHABLE);
        gc->gc.gc_refs = FROM_GC(gc)->ob_refcnt;
    }
}
```

接下来的动作就是要将环引用从引用中摘除。

```
[gcmodule.c]
static void subtract_refs(PyGC_Head *containers)
{
    traverseproc traverse;
    PyGC_Head *gc = containers->gc.gc_next;
    for (; gc != containers; gc=gc->gc.gc_next) {
        traverse = FROM_GC(gc)->ob_type->tp_traverse;
        (void) traverse(FROM_GC(gc),(visitproc)visit_decref, NULL);
    }
}
```

其中的 traverse 是与特定的 container 对象相关的，在 container 对象的类型对象中定义。一般来说，traverse 的动作都是遍历 container 对象中的每一个引用，然后对引用进行某种动作，而这个动作在 subtract_refs 中就是 visit_decref，它以一个回调函数的形式传递到 traverse 操作中。作为例子，我们来看看 PyDictObject 对象所定义的 traverse 操作。

```
[object.h]
typedef int (*visitproc)(PyObject *, void *);
typedef int (*traverseproc)(PyObject *, visitproc, void *);

[dictobject.c]
PyTypeObject PyDict_Type = {
    ……
    (traverseproc)dict_traverse,        /* tp_traverse */
    ……
};

static int dict_traverse(PyObject *op, visitproc visit, void *arg)
{
    int i = 0, err;
    PyObject *pk;
    PyObject *pv;

    while (PyDict_Next(op, &i, &pk, &pv)) {
        visit(pk, arg);
        visit(pv, arg);
    }
}
```

对于 dict 中的所有键和所有值都回调用回调函数，即 subtract_refs 中传递进来的 visit_decref。

```
[gcmodule.c]
static int visit_decref(PyObject *op, void *data)
{
    //PyObject_IS_GC 判断 op 指向的对象是不是被垃圾收集监控的
    //通常在 container 对象的 type 对象中有 Py_TPFLAGS_HAVE_GC 符号
    //标识 container 对象是被垃圾收集监控的
    if(PyObject_IS_GC(op)) {
    PyGC_Head *gc = AS_GC(op);
        if (gc->gc.gc_refs > 0)
```

```
        gc->gc.gc_refs--;
    }
    return 0;
}
```

在完成了 subtract_refs 之后，可收集对象链表中所有 contianer 对象之间的环引用都被摘除了。这时，有一些 container 对象的 PyGC_Head.gc.gc_ref 还不为 0，这就意味着存在对这些对象的外部引用，这些对象，就是开始标记——清除算法的 root object 集合。

图 16-17 展示了图 16-16 所示的例子在经过了 update_refs 和 subtract_refs 两步处理后所得到的 root object 集合。

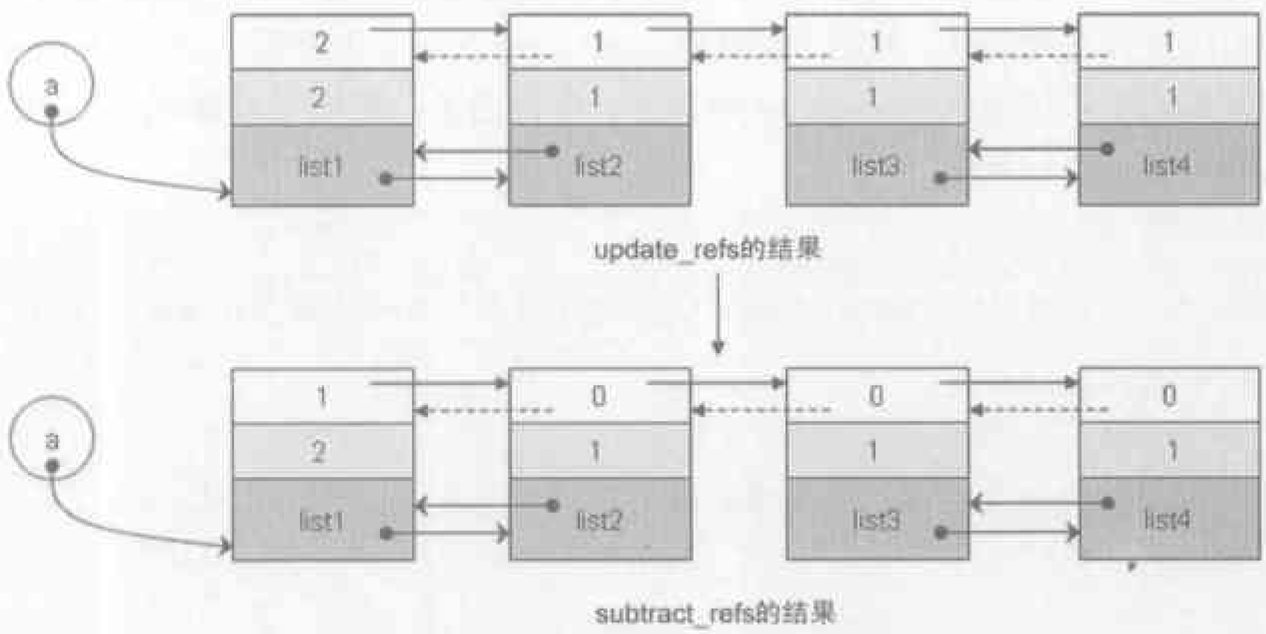



图 16-17 update_refs 和 subtract_refs 的执行结果

### 16.4.3.2 垃圾标记

成功地寻找到 root object 集合之后，我们就可以从 root object 出发，沿着引用链，一个接一个地标记不能回收的内存，由于 root object 集合中的对象是不能回收的，因此，被这些对象直接或间接引用的对象也是不能回收的。在从 root object 出发之前，我们首先要将现在的内存链表一分为二，一条链表中维护 root object 集合，成为 root 链表，而另一条链表中维护剩下的对象，称为 unreachable 链表。之所以要剖成两个链表，是基于这样的一种考虑：显然，现在的 unreachable 链表是名不副实的，其中可能存在被 root 链表中的对象直接或间接引用的对象，这些对象也是不能回收的，一旦在标记的过程中，发现了这样的对象，就将其从 unreachable 链表中移到 root 链表中；当完成标记后，unreachable 链表中剩下的对象就是名副其实的垃圾对象了，接下来的垃圾回收只需限制在 unreachable 链表中即可。

为此，Python 准备了一条名为 unreachable 的链表，通过 move_unreachable 完成对原始链表的剖分（见代码清单 16-9）。

代码清单 16-9

```
[gcmodule.c]
static void move_unreachable(PyGC_Head *young, PyGC_Head *unreachable)
{
    PyGC_Head *gc = young->gc.gc_next;
    while (gc != young) {
        PyGC_Head *next;
        //[1]: 对于root object，设置其gc_refs为GC_REACHABLE标志
        if (gc->gc.gc_refs) {
            PyObject *op = FROM_GC(gc);
            traverseproc traverse = op->ob_type->tp_traverse;
            gc->gc.gc_refs = GC_REACHABLE;
            (void) traverse(op, (visitproc)visit_reachable, (void *)young);
            next = gc->gc.gc_next;
        }
        //[2]: 对于非root对象，移到unreachable链表中，
        //并标记为GC_TENTATIVELY_UNREACHABLE
        else {
            next = gc->gc.gc_next;
            gc_list_move(gc, unreachable);
            gc->gc.gc_refs = GC_TENTATIVELY_UNREACHABLE;
        }
        gc = next;
    }
}

static int visit_reachable(PyObject *op, PyGC_Head *reachable)
{
    if (PyObject_IS_GC(op)) {
        PyGC_Head *gc = AS_GC(op);
        const int gc_refs = gc->gc.gc_refs;
        //[3]: 对于还没有处理的对象，恢复其gc_refs
        if (gc_refs == 0) {
            gc->gc.gc_refs = 1;
        }
        //[4]: 对于已经被挪到unreachable链表中的对象，将其再次挪到原来的链表
        else if (gc_refs == GC_TENTATIVELY_UNREACHABLE) {
            gc_list_move(gc, reachable);
            gc->gc.gc_refs = 1;
        }
         else {
            assert(gc_refs > 0 || gc_refs == GC_REACHABLE || gc_refs ==
              GC_UNTRACKED);
         }
    }
    return 0;
}
```

在 move_unreachable 中，沿着可收集对象链表依次向前，并检查其 PyGC_Head.gc.gc_ref 值，我们发现，这里的动作是遍历链表，而并非从 root object 集合出发，遍历引用链。这将导致一个微妙的结果，即当检查到一个 gc_ref 为 0 的对象时，我们并不能立即断定这个对象就是垃圾对象。因为这个对象之后的对象链表上，也许还会遇到一个

root object，而这个 root object 将引用该对象。所以，这个对象只是一个可能的垃圾对象，因此在代码清单 16-9 的[2]处将其暂时性地标注为 GC_TENTATIVELY_UNREACHABLE，但是还是通过 gc_list_move 将其搬移到了 unreachable 对象链表中，不过不要紧，马上我们就能看到，Python 留下了一条后路。

当在 move_unreachable 中遇到一个 gc_refs 不为 0 的对象 A 时，显然，A 是 root object 或从某个 root object 能引用到的对象，而 A 所引用的所有对象也都是不可回收的对象。因此，在代码清单 16-9 的[1]处，会再次调用与特定对象相关的 traverse 操作，依次对 A 中所引用的对象进行调用 visit_reachable。在 visit_reachable 的[4]处我们就可以发现，如果 A 所引用的对象之前曾被标注为 GC_TENTATIVELY_UNREACHABLE，那么现在能通过 A 访问到它，就意味着它也是一个不可回收对象，所以 Python 会重新将其从 unreachable 链表中搬移回原来的列表。注意，这里的 reachable，即是 move_ nreachabl 中的 young，也就是我们所谓的 root object 链表。Python 还会将其 gc_refs 设置为 1，表示该对象是一个不可回收的对象。同样，在[1]处，我们看到对 A 所引用的 gc_refs 为 0 的对象，也将其 gc_refs 设置为了 1。想一想，这样的对象是什么对象呢？显然，它是在链表中 move_unreachable 操作还没有访问到的对象，这样，Python 就直接掐断了之后 move_unreachable 访问它时将其移动到 unreachable 链表的诱因。图 16-18 显示了链表被剖分后的结果。

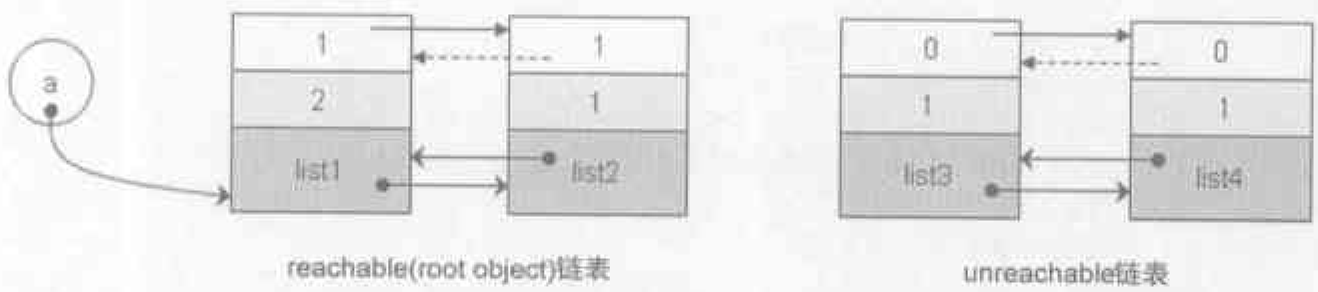



图 16-18 最终获得的 reachable 链表和 unreachable 链表

当 move_unreachable 完成之后，最初的一条链表就被剖分成了两条链表，在 unreachable 链表中，就是我们所发现的垃圾对象，是垃圾回收的目标。但是，等一等，在 unreachable 链表中，所有的对象都能被安全地回收吗？恐怕未必。当 20 世纪初人们以为物理学的大厦已经建立完毕时，这座大厦的上空又出现了小小的三朵乌云，而这些乌云最终将会把经典物理学的大厦摧枯拉朽般地摧毁。现在，我们也遇到了这样的乌云。

问题出在一种特殊的 container 对象，即从类对象实例化得到的实例对象。当我们用 Python 定义一个 class 时，可以为这个 class 定义一个特殊的方法：__del__，这在 Python 中被称为 finalizer。当一个拥有 finalizer 的实例对象被销毁时，首先会调用这个 finalizer，因为这个__del__就是 Python 为开发人员提供的在对象被销毁时进行某些资源释放的 Hook 机制。现在问题来了，我们已经知道，最终在 unreachable 链表中出现的对象都是

只存在循环引用的对象，需要被销毁。但是假如现在在 unreachbale 中，有两个对象，对象 B 在 finalizer 中调用了对象 A 的某个操作，这意味着安全的垃圾回收必须保证对象 A 一定要在对象 B 之后被回收，但是 Python 无法做到这一点，Python 在回收垃圾时不能保证回收的顺序。于是，有可能在 A 被销毁之后，B 在销毁时访问已经不存在的 A，毫无疑问，Python 遇到麻烦了。虽然同时满足存在 finalizer 和循环引用这两个条件的概率非常低，但 Python 不能对此置之不理。这是一个非常棘手的问题，从 Python 中拿掉__del__显然是很愚蠢的，所以 Python 采用了一种保守的做法，即将 unreachable 链表中的拥有 finalizer 的 PyInstanceObject 对象统统都移到一个名为 garbage 的 PyListObject 对象中。

### 16.4.3.3 垃圾回收

要回收 unreachable 链表中的垃圾对象，就必须先打破对象间的循环引用，前面我们已经详细地阐述了如何打破循环引用的算法。在寻找 root object 时，我们引入了 gc_refs 来模拟这个打破过程，现在我们要真刀真枪对 ob_refcnt 下手了，直到 unreachable 链表中的每一个对象的 ob_refcnt 变为 0，引发对象的销毁。

```
[gcmodule.c]
static int
gc_list_is_empty(PyGC_Head *list)
{
    return (list->gc.gc_next == list);
}

static void delete_garbage(PyGC_Head *collectable, PyGC_Head *old)
{
    inquiry clear;

    while (!gc_list_is_empty(collectable)) {
        PyGC_Head *gc = collectable->gc.gc_next;
        PyObject *op = FROM_GC(gc);

        if ((clear = op->ob_type->tp_clear) != NULL) {
            Py_INCREF(op);
            clear(op);
            Py_DECREF(op);
        }

        if (collectable->gc.gc_next == gc) {
            /* object is still alive, move it, it may die later */
            gc_list_move(gc, old);
            gc->gc.gc_refs = GC_REACHABLE;
        }
    }
}
```

其中会调用 container 对象的类型对象中的 tp_clear 操作，这个操作会调整

container 对象中每个引用所用的对象的引用计数值，从而完成打破循环的最终目标。作为一个例子，我们来看看 PyListObject 的 clear 操作。

```
[listobject.c]
static int list_clear(PyListObject *a)
{
    int i;
    PyObject **item = a->ob_item;
    if (item != NULL) {
        i = a->ob_size;
        a->ob_size = 0;
        a->ob_item = NULL;
        a->allocated = 0;
        while (--i >= 0) {
            Py_XDECREF(item[i]);
        }
        PyMem_FREE(item);
    }
    return 0;
}
```

我们注意到，在 delete_garbage 中，有一些 unreachable 链表中的对象会被重新送回到 reachable 链表(即 delete_garbase 函数的 old 参数)中，这是由于在进行 clear 动作时，如果成功进行，则通常一个对象会把自己从垃圾收集机制维护的链表中摘除（也就是这里的 collectable 链表）。由于某些原因，对象可能在 clear 动作时，没有成功完成必要的动作，从而没有将自己从 collectable 链表摘除，这表示对象认为自己还不能被销毁，所以 Python 需要将这种对象放回 reachable 链表中。

我们来看看图 16-18 所示的 unreachable 链表中的 list3 和 list4 是如何被回收的。在 delete_garbage 中，假如首先处理 list3，调用其 list_clear，那么会减少 list4 的引用计数，这将导致 list4 的 ob_refcnt 为 0，引发对象销毁动作，会调用 list4 的 list_dealloc。

```
[listobjec.c]
static void list_dealloc(PyListObject *op)
{
    int i;
    PyObject_GC_UnTrack(op);
    if (op->ob_item != NULL) {
        i = op->ob_size;
        while (--i >= 0) {
            Py_XDECREF(op->ob_item[i]);
        }
        PyMem_FREE(op->ob_item);
    }
    ……
}
```

首先会将 list4 从可收集对象链表中摘除，然后如同 list_clear 所作的，会调整 list4 所引用的所有对象的引用计数，这个动作立即就影响到了 list3，并使其引用计数

变为 0，同样，list3 的销毁动作也被触发了。如此一来，list3 和 list4 就都被安全地回收了。

## 16.4.4 垃圾收集全景

到此，我们已经详细地剖析了 Python 中垃圾收集机制的所有细节及隐秘之处，作为这些细节的综合，是时候来看一看 Python 中那个实际完成垃圾收集的 collect 是如何实现的。了解了 collect，就功德圆满了。

```
[gcmodule.c]
static long collect(int generation)
{
    int i;
    long m = 0; /* # objects collected */
    long n = 0; /* # unreachable objects that couldn't be collected */
    PyGC_Head *young; /* the generation we are examining */
    PyGC_Head *old; /* next older generation */
    PyGC_Head unreachable; /* non-problematic unreachable trash */
    PyGC_Head finalizers;  /* objects with, & reachable from, __del__ */
    PyGC_Head *gc;

    if (delstr == NULL) {
        delstr = PyString_InternFromString("__del__");
        if (delstr == NULL)
            Py_FatalError("gc couldn't allocate \"__del__\"");
    }

    /* update collection and allocation counters */
    if (generation+1 < NUM_GENERATIONS)
        generations[generation+1].count += 1;
    for (i = 0; i <= generation; i++)
        generations[i].count = 0;

    /* merge younger generations with one we are currently collecting */
    /* 将比当前处理的"代"更年轻的"代"的链表合并到当前"代"中
    for (i = 0; i < generation; i++) {
        gc_list_merge(GEN_HEAD(i), GEN_HEAD(generation));
    }

    /* handy references */
    young = GEN_HEAD(generation);
    if (generation < NUM_GENERATIONS-1)
        old = GEN_HEAD(generation+1);
    else
        old = young;

    // 在待处理链表上进行打破循环的模拟，寻找 root object
    update_refs(young);
    subtract_refs(young);

    //将待处理链表中的 unreachable object 转移到 unreachable 链表中
```

```
    //处理完成后，当前"代"中只剩下 reachable object 了
    gc_list_init(&unreachable);
    move_unreachable(young, &unreachable);

    //如果可能，将当前"代"中的 reachable object 合并到更老的"代"中
    if (young != old)
        gc_list_merge(young, old);

    //对于 unreachable 链表中的对象，如果其带有__del__函数，则不能安全回收
    //需要将这些对象收集到 finalizers 链表中，因此，这些对象引用的对象也不能
    //回收，也需要放入 finalizers 链表中
    gc_list_init(&finalizers);
    move_finalizers(&unreachable, &finalizers);
    move_finalizer_reachable(&finalizers);

    //处理弱引用（weakref），如果可能，调用弱引用中注册的 callback 操作
    handle_weakrefs(&unreachable, old);

    //对 unreachable 链表上的对象进行垃圾回收操作
    delete_garbage(&unreachable, old);

    //将含有__del__操作的实例对象收集到 Python 内部维护的名为 garbage 的链表中
    //同时将 finalizers 链表中所有对象加入 old 链表中
    (void)handle_finalizers(&finalizers, old);
    ......
}
```

我们注意到在 collect 函数中，还有对 Python 中弱引用（weakref）的处理，因为 weakref 能够注册 callback 操作，所以这个行为有点类似带有__del__的实例对象。但是它们还是有本质的不同，weakref 能够被正确地清理掉，虽然必须引入一些额外的繁琐的操作，这些操作就隐身在 handle_weakrefs 中。而带有__del__的实例对象是不能自动被清除的，最终将被放入 garbage 链表中。对于弱引用的处理，这里就不深入了，有兴趣的读者可以自己参考 Python 源码。

到了这里，我们需要指出一点，Python 的垃圾收集机制完全是为了处理循环引用而设计的，虽然几乎大多数对象在创建时都会通过 PyObject_GC_New，并最终调用_PyObject_GC_New，将创建的对象纳入垃圾收集机制的监控中。但是有趣的是，被垃圾收集监控的对象并非只有垃圾收集机制才能回收，正常的引用计数就能销毁一个被纳入垃圾收集机制监控的对象。比如我们来看看 PyFunction 对象的正常销毁。

```
[funcobject.c]
static void func_dealloc(PyFunctionObject *op)
{
    _PyObject_GC_UNTRACK(op);
    ......
    PyObject_GC_Del(op);
}

[gcmodule.c]
```

```
void PyObject_GC_Del(void *op)
{
    PyGC_Head *g = AS_GC(op);
    if (IS_TRACKED(op))
        gc_list_remove(g);
    if (generations[0].count > 0) {
        generations[0].count--;
    }
    PyObject_FREE(g); //最终调用 PyObject_Free
}
```

如果 PyFunctionObject 对象因为正常的引用计数维护到达引用计数为 0 的状态，就会调用 func_dealloc。我们看到，PyFunctionObject 对象主动将自己从垃圾收集监控的链表中摘除，然后调用 PyObject_GC_Del 释放内存，之所以需要调用 PyObject_GC_Del，主要是为了将指向 PyObject 的指针调整为指向 PyGC_Head 的指针，以释放正确的内存。

所以，虽然有很多对象挂在垃圾收集机制监控的链表上，但实际上更多时候，是引用计数机制在维护这些对象，只有对引用计数无能为力的循环引用，垃圾收集机制才会起作用。事实上，对循环引用之外的对象，垃圾收集是无能为力的。因为挂在垃圾收集机制上的对象都是引用计数不为 0 的，因为如果为 0，早就被引用计数机制“干掉”了。而引用计数不为 0 的对象只有两种情况：一是被程序使用的对象；二是循环引用中的对象。被程序使用的对象是不能被回收的，所以垃圾回收能且只能处理循环引用中的对象。

另一点需要说明的是，PyObject_GC_New 底层是以我们之前剖析的 PyObject_Malloc 作为真正申请内存的接口的，这意味着在大多数情况下，Python 都在使用内存池。本书中我们剖析过的最大的对象就是 PyTypeObject，而这个对象也不过 200 个字节，小于 256 个字节，同样可以使用内存池。所以我们可以将垃圾收集和内存管理完全融为一体了。

## 16.4.5 Python 中的 gc 模块

Python 中通过 gc 模块为程序员提供了观察和手动使用 gc 机制的接口，这一节，我们通过 gc 模块进行一些观察，以加深对垃圾收集机制的理解。本节不对 gc 模块的使用进行介绍，关于 gc 模块的使用，请参考 Python 文档。

```
[gc1.py]
import gc
class A(object):
    pass

class B(object):
    pass

gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK)
a = A()
b = B()
```

```
del a
del b
gc.collect()

结果:
F:\PythonBook\Src\memory>python gcl.py
gc: collecting generation 2...
gc: objects in each generation: 341 2692 0
gc: done.
gc: collecting generation 2...
gc: objects in each generation: 0 0 3027
gc: done.
```

通过 gc1.py 的演示，证明了对于引用计数机制能正常维护的对象，垃圾收集确实起不到任何作用。

```
[gc2.py]
import gc
class A(object):
   pass

class B(object):
   pass

gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK)
a = A()
b = B()
a.b = b
b.a = a
del a
del b
gc.collect()

运行结果:
F:\PythonBook\Src\memory>python gc2.py
gc: collecting generation 2...
gc: objects in each generation: 345 2692 0
gc: collectable <A 00B46850>
gc: 0.0160s elapsed.
gc: collectable <B 00B46870>
gc: 1201346676.5930s elapsed.
gc: collectable <dict 00B48300>
gc: 0.0160s elapsed.
gc: collectable <dict 00B48270>
gc: 1201346676.5930s elapsed.
gc: done, 4 unreachable, 0 uncollectable.
gc: collecting generation 2...
gc: objects in each generation: 0 0 3031
gc: done.
```

正如 gc2.py 的运行结果显示的，当存在循环引用时，引用计数确实不起作用了，而垃圾收集则能正确回收内存。

```
[gc3.py]
import gc
```

```
class A(object):
    def __del__(self):
        pass

class B(object):
    def __del__(self):
        pass

gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK)
a = A()
b = B()
a.b = b
b.a = a
del a
del b
gc.collect()

运行结果:
F:\PythonBook\Src\memory>python gc3.py
gc: collecting generation 2...
gc: objects in each generation: 353 2692 0
gc: uncollectable <A 00B46990>
gc: uncollectable <B 00B469B0>
gc: uncollectable <dict 00B43E40>
gc: uncollectable <dict 00B481E0>
gc: done, 4 unreachable, 4 uncollectable.
gc: collecting generation 2...
gc: objects in each generation: 0 0 3037
gc: done.
```

由于我们执意在类对象中添加了`__del__`操作，所以 GC 很生气，后果很严重。不管是对象 a 本身（需要注意，显示结果中的 A 不是说 A 不能回收，而是**类型为** A 的 a 不能回收），就连 a 对象内维护的`__dict__`也不能回收。真的是非常严重的后果，这不就是内存泄漏么？所以，没什么事，千万不要轻易启用`__del__`操作。

### 16.4.6 总结

尽管 Python 采用了最经典的（换句话说，最土的）引用计数来作为自动内存管理的方案，但是 Python 采用了多种方式来弥补引用计数的不足，内存池的大量使用，标记-清除垃圾收集技术的使用都极大地完善了 Python 的内存管理机制。尽管引用计数还存在着需要花费额外的内存维护引用计数值的毛病，但现在已经是 2008 年了，已经不是 64KB 的年代了，这点花销，我想，我们还是支付得起的。而另一方面，引用计数也有其优点，倘若它没有优点，早就被扫进历史的垃圾堆了。比如，引用计数将垃圾收集的开销分摊在了整个运行时，这对于 Python 的响应性能是非常有好处的。又比如，当前的研究表明，

在分布式环境下，引用计数是目前最有效的垃圾收集技术。当然，这已经不是我们需要关心的话题了。

内存管理和垃圾收集，是一门非常精细和繁琐的技术，本章进行的剖析无法覆盖 Python 内存管理机制的所有细微之处，如果你不满足于本章的剖析，那么请打开你的代码浏览工具，在本章的基础上继续深入。

［Ｇｅｎｅｒａｌ　Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝Ｐｙｔｈｏｎ源码剖析：深度探索动态语言核心技术
ＳＳ号＝１２００８２１８
作者＝陈儒著
页数＝４８０
出版社＝电子工业出版社
出版日期＝２００８．６